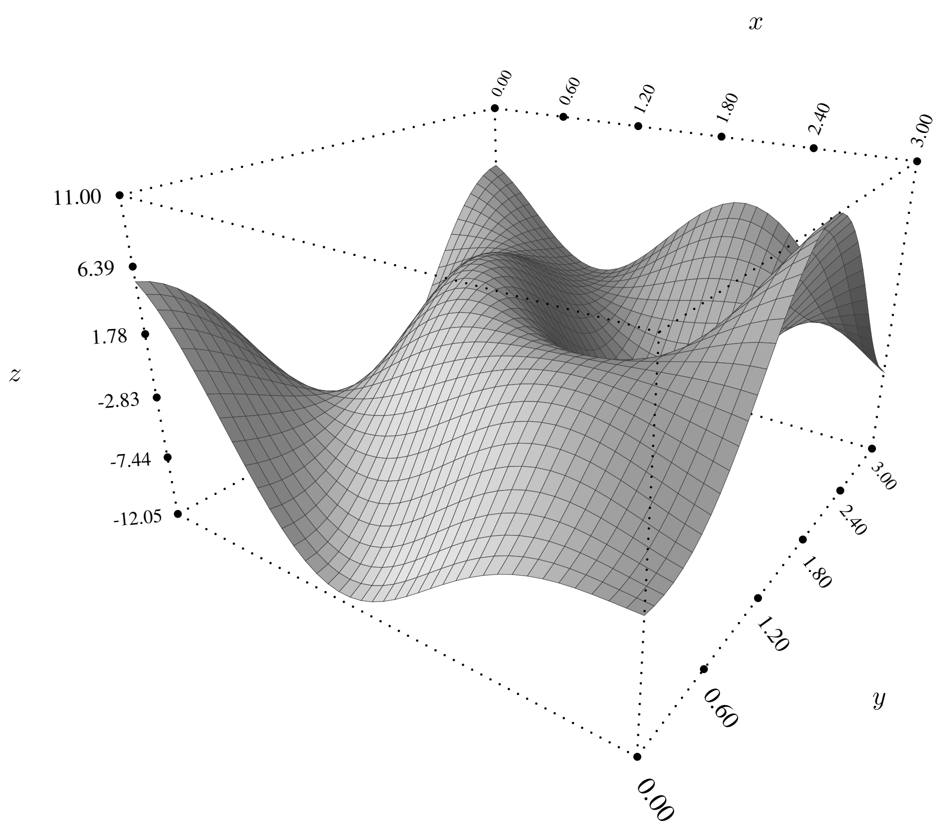
Notational innovations for rapid application development



Dennis Furey
`ursala-users@freelists.org`

November 19, 2012

Abstract

This manual introduces and comprehensively documents a style of software prototyping and development involving a novel programming language. The language draws heavily on the functional paradigm but lies outside the mainstream of the subject, being essentially untyped and variable free. It is based on a firm semantic foundation derived from a well documented virtual machine model visible to the programmer. Use of a concrete virtual machine promotes segregation of procedural considerations within a primarily declarative formalism.

Practical advantages of the language are a simple and unified interface to several high performance third party numerical libraries in C and Fortran, a convenient mechanism for unrestricted client/server interaction with local or remote command line interpreters, built in support for high quality random variate generation, and an open source compiler with an orthogonal, table driven organization amenable to user defined enhancements.

This material is most likely to benefit mathematically proficient software developers, scientists, and engineers, who are arguably less well served by the verbose and restrictive conventions that have become a fixture of modern programming languages. The implications for generality and expressiveness are demonstrated within.

Contents

Part I

Introduction

Concurrently while your first question may be the most pertinent, you may or may not realize it is also the most irrelevant.

The Architect in *The Matrix Reloaded*

1 Motivation

Who needs another programming language? The very idea is likely to evoke a frosty reception in some circles, justifiably so if its proponents are insufficiently appreciative of a simple economic fact. The most expensive thing about software is the cost of customizing or maintaining it, including the costs of training or recruitment of suitably qualified individuals. These costs escalate in the case of esoteric software technologies, of which unconventional languages are the prime example, and they ordinarily will take precedence over other considerations.

1.1 Intended audience

While there is no compelling argument for general commercial deployment of the tools and techniques described in this manual, there is nevertheless a good reason for them to exist. Many so called mature technologies from which organizations now benefit handsomely began as research projects, without which all progress comes to a standstill. Furthermore, this material may be of use to the following constituencies of early adopters.

1.1.1 Academic researchers

Perhaps you've promised a lot in your thesis proposal or grant application and are now wondering how you'll find an extra year or two for writing the code to support your claims. Outsourcing it is probably not an option, not just because of the money, but because the ideas are too new for anyone but you and a few colleagues to understand. Textbook software engineering methodologies can promise no improvement in productivity because the exploratory nature of the work precludes detailed planning. Automated code generation tools address only the user interface rather than the substance of the application.

The language described in this manual provides you with a path from rough ideas to working prototypes in record time. It does so by keeping the focus on a high level of abstraction that dispenses with the tedium and repetition perceived to a greater degree in other languages. By a conservative estimate, you'll write about one tenth the number of lines of code in this language as in C or Java to get the same job done.[1]

How could such a technology exist without being more widely known? The deal breaker for a commercial organization would be the cost of retraining, and the risk of something untried. These issues pose no obstacle to you because learning and evaluating new ideas is your bread and butter, and financially you have nothing to lose.

1.1.2 Hackers and hobbyists

This group merits pride of place as the source of almost every significant advance in the history of computing. A reader who believes that stretching the imagination and looking for new ways of thinking are ends in themselves will find something of value in these pages.

The functional programming community has changed considerably since the `lisp` era, not necessarily for the better unless one accepts the premise of the compiler writer as policy maker. We are now hard pressed to find current research activity in the field that is not concerned directly or indirectly with type checking and enforcement.

The subject matter of this document offers a glimpse of how functional programming might have progressed in the absence of this constraint. Not too surprisingly, we find ever more imaginative and ubiquitous use of higher order functions than is conceivable within the confines of a static type discipline.

1.1.3 Numerical analysts

Perhaps you have no great love for programming paradigms, but you have a real problem to solve that involves some serious number crunching. You will already be well aware of many high quality free numerical libraries, such as `lapack`, `Kinsol`, `fftw`, `gsl`, *etcetera*, which are a good start, but you don't relish the prospect of writing hundreds of lines of glue code to get them all to work together. Maybe on top of that you'd like to leverage some existing code written in mutually incompatible domain specific languages that has no documented API at all but is invoked by a command line interpreter such as `Octave` or `R` or their proprietary equivalents.

This language takes about a dozen of the best free numerical libraries and not only combines them into a consistent environment, but simplifies the calling conventions to the extent of eliminating anything pertaining to memory management or mutable storage. The developer can feed the output from one library function seamlessly to another even if the libraries were written in different languages. Furthermore, any command line interpreter present on the host system can be invoked and controlled by a function call from within the language, with a transcript of the interaction returned as the result.

[1] I'm a big fan of C, as all real programmers are, but I still wouldn't want to use it for anything too complicated.

1.1.4 Independent consultants

Commercial use of this technology may be feasible under certain circumstances. One could envision a sole proprietorship or a small team of academically minded developers, building software for use in house, subject to the assumption that it will be maintained only by its authors. Alternatively, there would need to be a commitment to recruit for premium skills.

Possible advantages in a commercial setting are rapid adaptation to changing requirements or market conditions, for example in an engineering or trading environment, and fast turnaround in a service business where software is the enabling technology. A less readily quantifiable benefit would be the long term effects of more attractive working conditions for developers with a preference for advanced tools.

1.2 Grand tour

The remainder of this chapter attempts to convey a flavor for the kinds of things that can be done well with this language. Examples from a variety of application areas are presented with explanations of the main points. These examples are not meant to be fully comprehensible on a first reading, or else the rest of the manual would be superfluous. Rather, they are intended to allow readers to make an informed decision as to whether the language would be helpful enough to be worth learning.

1.2.1 Graph transformation

This example is a type of problem that occurs frequently in CAD applications. Given a model for a system, we seek a simpler model if possible that has the same externally observable behavior. If the model represents a circuit to be synthesized, the optimized version is likely to be conducive to a smaller, faster circuit.

Theory

A graph such as the one shown in Figure 1.1 represents a system that interacts with its environment by way of input and output signals. For concreteness, we can imagine the inputs as buttons and the outputs as lights, each identified with a unique label. When an acceptable combination of buttons is pressed, the system changes from its present state to another designated state, and in so doing emits signals on the required outputs.

This diagram summarizes everything there is to know about the system according to the following conventions.

- Each circle in the diagram represents a state.
- Each arrow (or "transition") represents a possible change of state, and is drawn connecting a state to its successor with respect to the change.

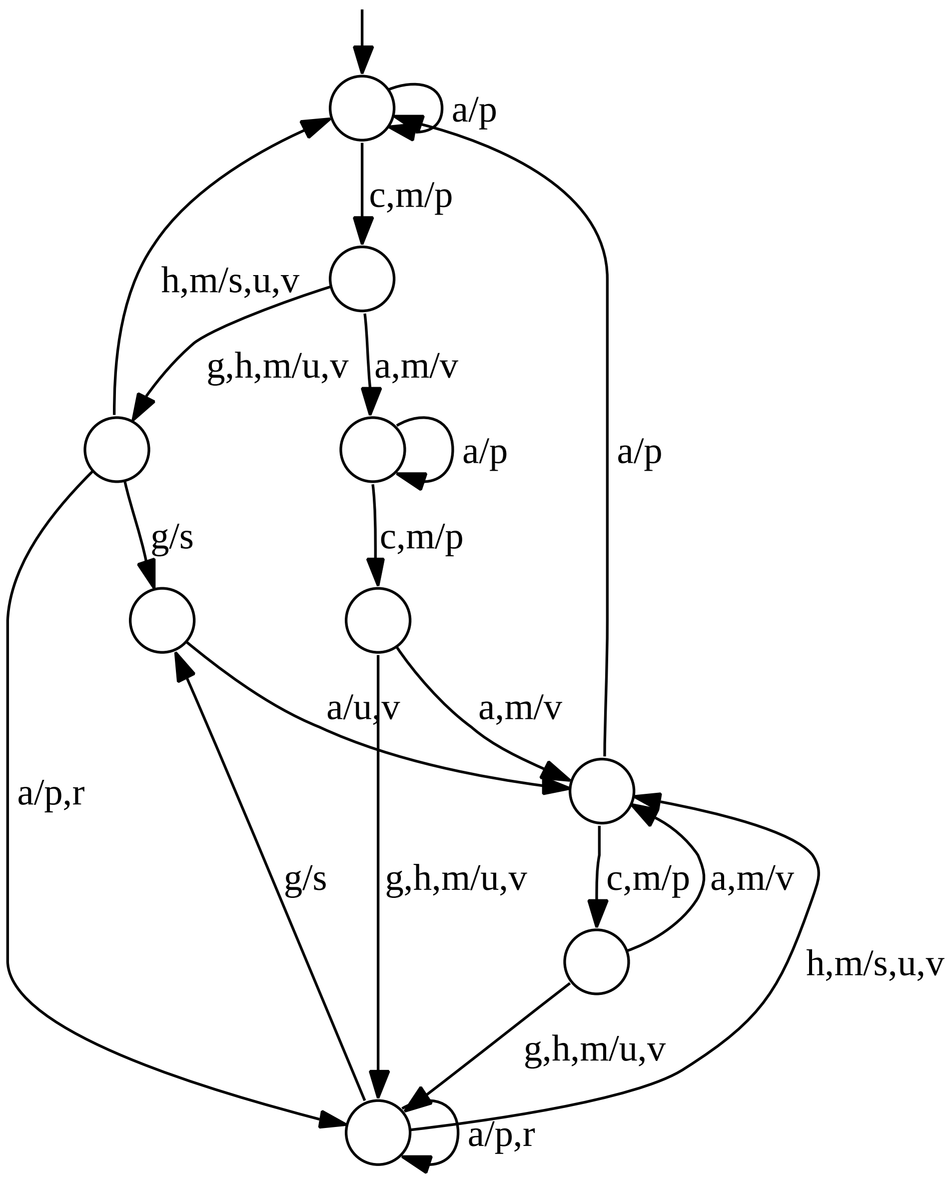


Figure 1.1: a finite state transducer

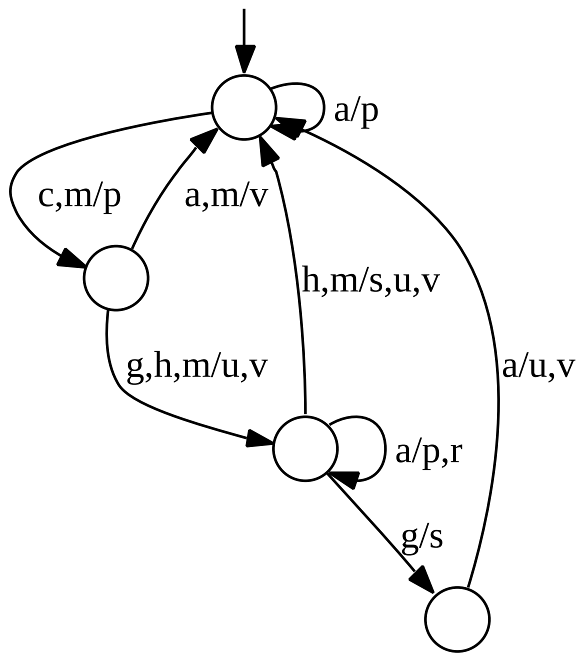


Figure 1.2: a smaller equivalent version

- Each transition is labeled with a set of input signal names, followed by a slash, followed by a set of output signal names.
  - The input signal names labeling a transition refer to the inputs that cause it to happen when the system is in the state where it originates.
  - The output signal names labeling a transition refer to the outputs that are emitted when it happens.
- An unlabeled arrow points to the initial state.

Problem statement

Two systems are considered equivalent if their observable behavior is the same in all circumstances. The state of a system is considered unobservable. Only the input and output protocol is of interest. We can now state the problem as follows:

Using whatever data structure you prefer, implement an algorithm that transforms a given system specification to a simpler equivalent one if possible.

For example, the system shown in Figure 1.1 could be transformed to the one in Figure 1.2, because both have the same observable behavior, but the latter is simpler because it has only four states rather than nine.

Listing 1.1 concrete representation of the system in Figure 1.1

```
#binary+

sys =

{
   0: {({'a'},{'p'}): 0,({'c','m'},{'p'}): 7},
   8: {({'a'},{'p'}): 0,({'c','m'},{'p'}): 2},
   4: {
      ({'a'},{'p','r'}): 9,
      ({'g'},{'s'}): 3,
      ({'h','m'},{'s','u','v'}): 0},
   2: {
      ({'a','m'},{'v'}): 8,
      ({'g','h','m'},{'u','v'}): 9},
   6: {({'a'},{'p'}): 6,({'c','m'},{'p'}): 1},
   1: {
      ({'a','m'},{'v'}): 8,
      ({'g','h','m'},{'u','v'}): 9},
   9: {
      ({'a'},{'p','r'}): 9,
      ({'g'},{'s'}): 3,
      ({'h','m'},{'s','u','v'}): 8},
   3: {({'a'},{'u','v'}): 8},
   7: {
      ({'a','m'},{'v'}): 6,
      ({'g','h','m'},{'u','v'}): 4}}
```

Data structure

A simple, intuitive data structure is perfectly serviceable for this example.

- A character string is used for each signal name, a set of them for each set thereof, and a pair of sets of character strings to label each transition.
- For ease of reference, each state is identified with a unique natural number, with 0 reserved for the initial state.
- A transition is represented by its label and its associated destination state number.
- A state is fully characterized by its number and its set of outgoing transitions.
- The entire system is represented by the set of the representations of its states.

The language uses standard mathematical notation of braces and parentheses enclosing comma separated sequences for sets and tuples, respectively. A colon separated pair is an alternative notation optionally used in the language to indicate an association or assignment, as in `x: y`. White space is significant in this notation and it denotes a purely non-mutable, compile-time association.

Listing 1.2 optimization algorithm

```
#import std
#import nat

#library+

optimized =

|=&mnS; -+
   ^Hs\~&hS *+ ^|^(~&,*+ ^|/~&)+ -:+ *= ~&nS; ^DrlXS/nleq$- ~&,
   ^= ^H\~& *=+ |=+ ==++ ~~bm+ *mS+ -:+ ~&nSiiDPSLrlXS+-
```

Some test data of the required type are prepared as shown in Listing 1.1 in a file named `sys.fun`. (This source file suffix is standard.) The compiler will parse and evaluate such an expression with no type declaration required, although one will be used later to cast the binary representation for display purposes.

For the moment, the specification is compiled and stored for future use in binary form by the command

```
$ fun sys.fun
fun: writing `sys'
```

The command to invoke the compiler is `fun`. The dollar sign at the beginning of a line represents the shell command prompt throughout this manual. Writing the file `sys` is the effect of the `#binary+` compiler directive shown in the source. The file is named after the identifier with which the structure is declared.

Algorithm

In abstract terms, the optimization algorithm is as follows.

- Partition the set of states initially by equality of outgoing transition labels (ignoring their destination states).
- Further partition each equivalence class thus obtained by equivalence of transition termini under the relation implied hitherto.
- Iterate the previous step until a fixed point is reached.
- Delete all but one state from each terminal equivalence class, (with preference to the initial state where applicable) rerouting incident transitions on deleted states to the surviving class member as needed.

The entire program to implement this algorithm is shown in Listing 1.2. Some commentary follows, but first a demonstration is in order. To compile the code, we execute

```
$ fun cad.fun
fun: writing `cad.avm'
```

assuming that the source code in Listing 1.2 is in a file called `cad.fun`. The virtual machine code for the optimization function is written to a library file with suffix `.avm` because of the `#library+` compiler directive, rather than as a free standing executable.

Using the test data previously prepared, we can test the library function easily from the command line without having to write a separate driver.

```
$ fun cad sys --main="optimized sys" --cast %nsSWnASAS
{
   0: {({'a'},{'p'}): 0,({'c','m'},{'p'}): 1},
   4: {
      ({'a'},{'p','r'}): 4,
      ({'g'},{'s'}): 3,
      ({'h','m'},{'s','u','v'}): 0},
   1: {
      ({'a','m'},{'v'}): 0,
      ({'g','h','m'},{'u','v'}): 4},
   3: {({'a'},{'u','v'}): 0}}
```

This invocation of the compiler takes the library file `cad.avm`, with the suffix inferred, and the data file `sys` as command line arguments. The compiler evaluates an expression on the fly given in the parameter to the `--main` option, and displays its value cast to the type given by a type expression in the parameter to the `--cast` option. The result is an optimized version of the specification in Listing 1.1 as computed by the library function, displayed as an instance of the same type. This result corresponds to Figure 1.2, as required.

Highlights of this example

This example has been chosen to evoke one of two reactions from the reader. Starting from an abstract idea for a fairly sophisticated, non-obvious algorithm of plausibly practical interest, we've done the closest thing possible to pulling a working implementation out of thin air in three lines of code. However, it would be an understatement to say the code is difficult to read. One might therefore react either with aversion to such a notation because of its unfamiliarity, or with a sense of discovery and wonder at its extraordinary expressive power. Of course, the latter is preferable, but at least no time has been wasted otherwise. The following technical points are relevant for the intrepid reader wishing to continue.

Type expressions such as the parameter to the `--cast` command line option above, are built from a selection of primitive types and constructors each represented by a single letter combined in a postorder notation. The type `n` is for natural numbers, and `s` is for character strings. `S` is the set constructor, and `W` the constructor for a pair of the same type. Hence, `sS` refers to sets of strings, and `sSW` to pairs of sets of strings. The binary constructor `A` pertains to assignments. Type expressions are first class objects in the language and can be given symbolic names.

Pointer expressions such as `~&nSiiDPSLrlXS` from Listing 1.2, are a computationally universal language within a language using a postorder notation similar to type expressions as a shorthand for a great variety of frequently occurring patterns. Often they pertain to list or set transformations. They can be understood in terms of a well documented virtual machine code semantics, seen here in a more `lisp`-like notation, that is always readily available for inspection.

```
$ fun --main="~&nSiiDPSLrlXS" --decompile
main = compose(
   map field((0,&),(&,0)),
   compose(
      reduce(cat,0),
      map compose(
         distribute,
         compose(field(&,&),map field(&,0)))))
```

Library functions are reusable code fragments either packaged with the compiler or user defined and compiled into library files with a suffix of `.avm`. The function in this example is defined mostly in terms of language primitives except for one library function, `nleq`, the partial order relational predicate on natural numbers imported from the `nat` library. Functions declared in libraries are made accessible by the `#import` compiler directive.

Operators are used extensively in the language to express functional combining forms. The most frequently used operators are `+`, for functional composition, as in an expression of the form `f+ g`, and `;`, as in `g; f`, similar to composition with the order reversed. Another kind of operator is function application, expressed by juxtaposition of two expressions separated by white space. Semantically we have an identity `(f+ g) x` = `(g; f) x` = `f (g x)`, or simply `f g x`, as function application in this language is right associative.

Higher order functions find a natural expression in terms of operators. It is convenient to regard most operators as having binary, unary, and parameterless forms, so that an expression such as `g;` is meaningful by itself without a right operand. If `g;` is directly applied to a function `f`, we have the resulting function `g; f`. Alternatively, it would be meaningful to compose `g;` with a function `h`, where `h` is a function returning a function, as in `g;+ h`. This expression denotes a function returning a function similar to the one that would be returned by `h` with the added feature of `g` included in the result as a preprocessor, so to speak. Several cases of this usage occur in Listing 1.2.

Combining forms are associated with a rich variety of other operators, some of which are used in this example. Without detailing their exact semantics, we conclude this section with an informal summary of a few of the more interesting ones.

- The partition combinator, `|=`, takes a function computing an equivalence relation to the function that splits a list or a set into equivalence classes.
- The limit combinator, `^=`, iterates a function until a fixed point is reached.
- The fan combinator, `~~`, takes a function to one that operates on a pair by applying the given function to both sides.
- The reification combinator, `-:`, takes a finite set of pairs of inputs and outputs to the partial function defined by them.
- The minimization operator `$-`, takes a function computing a relational predicate to one that returns the minimum item of a list or set with respect to it.
- Another form of functional composition, `-+...+-`, constructs the composition of an enclosed comma separated sequence of functions.
- The binary to unary combinators `/` and `\` fix one side of the argument to a function operating on a pair. `f/k y` = `f(k,y)` and `f\k x` = `f(x,k)`, where it should be noted as usual that the expression `f/k` is meaningful by itself and consistent with this interpretation.

1.2.2 Data visualization

This example demonstrates using the language to manipulate and depict numerical data that might emerge from experimental or theoretical investigations.

Theory

The starting point is a quantity that is not known with certainty, but for which someone purports to have a vague idea. To be less vague, the person making the claim draws a bell shaped curve over the range of possible values and asserts that the unknown value is likely to be somewhere near the peak. A tall, narrow peak leaves less room for doubt than one that's low and spread out.[2]

Let us now suppose that the quantity is time varying, and that its long term future values are more difficult to predict than its short term values. Undeterred, we wish to construct a family of bell shaped curves, with one for each instant of time in the future. Because the quantity is becoming less certain, the long term future curves will have low, spread out peaks. However, we venture to make one mildly predictive statement, which is that the quantity is non-negative and generally follows an increasing trend. The peaks of the curves will therefore become laterally displaced in addition to being flatter.

It is possible to be astonishingly precise about being vague, and a well studied model for exactly the situation described has been derived rigorously from simple assumptions. Its essential features are as follows.

[2]apologies to those who might take issue with this greatly simplified introduction to statistics

A measure $\bar{x}$ of the expected value of the estimate (if we had to pick one), and its dispersion v are given as functions of time by these equations,

$$\begin{aligned}\bar{x}(t) &= me^{\mu t}\\ v(t) &= m^2 e^{2\mu t}\left(e^{\sigma^2 t}-1\right)\end{aligned}$$

where the parameters m, μ and σ are fixed or empirically determined constants. A couple of other time varying quantities that defy simple intuitive explanations are also defined.

$$\begin{aligned}\theta(t) &= \ln\left(\bar{x}(t)^2\right)-\frac{1}{2}\ln\left(\bar{x}(t)^2+v(t)\right)\\ \lambda(t) &= \sqrt{\ln\left(1+\frac{v(t)}{\bar{x}(t)^2}\right)}\end{aligned}$$

These combine to form the following specification for the bell shaped curves, also known as probability density functions.

$$(\rho(t))(x) = \frac{1}{\sqrt{2\pi}\lambda(t)x}\exp\left(-\frac{1}{2}\left(\frac{\ln x-\theta(t)}{\lambda(t)}\right)^2\right)$$

Whereas it would be fortunate indeed to find a specification of this form in a statistical reference, functional programmers by force of habit will take care to express it as shown if this is the intent. We regard ρ as a second order function, to which one plugs in a time value t, whereupon it returns another (unnamed) function as a result. This latter function takes a value x to its probability density at the given time, yielding the bell shaped curve when sampled over a range of x values.[3]

Problem statement

This problem is just a matter of muscle flexing compared to the previous one. It consists of the following task.

Get some numbers out of this model and verify that the curves look the way they should.

Surface renderings

A favorite choice for book covers and poster presentations is to render a function of two variables in an eye catching graphic as a three dimensional surface. A library for that purpose is packaged with the compiler. It features realistic shading and perspective from multiple views, and generates readable LaTeX code suitable for inclusion in documents or slides. Postscript and PDF renderings, while not directly supported, can be obtained through LaTeX for users of other document preparation systems.

The code to invoke the rendering library function for this model is shown in Listing 1.3 and the result in Figure 1.3. Assuming the code is stored in a file named `viz.fun`, it is compiled as follows.

[3]Some authors will use a more idiomatic notation like $\rho(x;t)$ to suggest a second order function, but seldom use it consistently.

Listing 1.3 code to generate the rendering in Figure 1.3

```
#import std
#import nat
#import flo
#import plo
#import ren
---------------------------- constants --------------------------------

imean   = 100.  # mean at time 0
sigma   = 0.3   # larger numbers make the variance increase faster
mu      = 0.6   # larger numbers make the mean drift upward faster

------------------------ functions of time ----------------------------

expectation = times/imean+ exp+ times/mu
theta       = minus^(ln+ ~&l,div\2.+ ln+ plus)^/sqr+expectation marv
lambda      = sqrt+ ln+ plus/1.+ div^/marv sqr+ expectation

marv = # variance of the marginal distribution

times/sqr(imean)+ times^(
   exp+ times/2.+ times/mu,
   minus\1.+ exp+ //times sqr sigma)

rho = # takes a positive time value to a probability density function

"t". 0.?=/0.! "x". div(
   exp negative div\2. sqr div(minus/ln"x" theta "t",lambda "t"),
   times/sqrt(times/2. pi) times/lambda"t" "x")

------------------------- image specifications  -----------------------
#binary+
#output dot'tex' //rendering ('ihn+',1.5,1.)

spread =

visualization[
   margin: 35.,
   headroom: 25.,
   picture_frame: ((350.,350.),(-15.,-25.)),
   pegaxis: axis[variable: '\textsl{time}'],
   abscissa: axis[variable: '\textsl{estimate}'],
   ordinates: <
      axis[variable: '$\rho$',hatches: ari5/0. .04,alias: (10.,0.)]>,
   curves: ~&H(
      * curve$[peg: ~&hr,points: * ^/~&l ^H\~&l rho+ ~&r],
      |=&r ~&K0 (ari41/75. 175.,ari31/0.1 .6))]
```

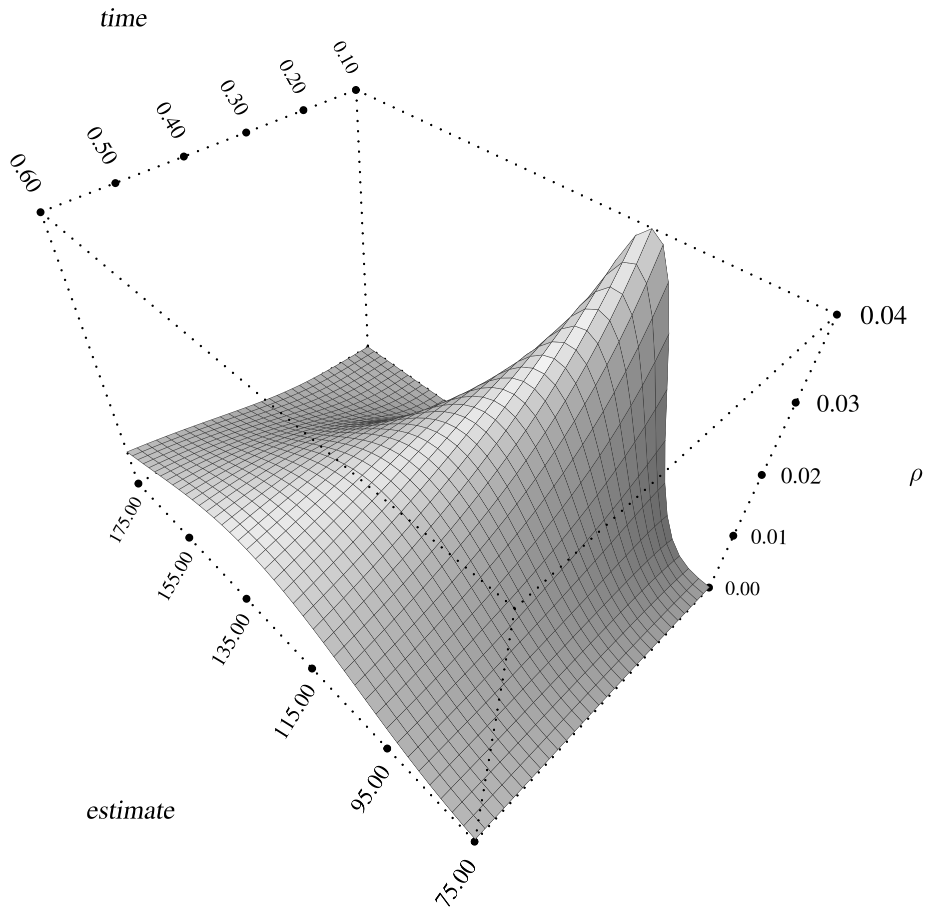


Figure 1.3: Probability density drifts and disperses with time as the estimate grows increasingly uncertain

```
$ fun flo plo ren viz.fun
fun: writing `spread'
fun: writing `spread.tex'
```

The output files in LaTeX and binary form are generated immediately at compile time, without the need to build any intermediate libraries or executables, because this application is meant to be used once only. This behavior is specified by the `#binary+` and `#output` compiler directives.

The main points of interest raised by this example relate to the handling of numerical functions and abstract data types.

Arithmetic operators are designated by alphanumeric identifiers such as `times` and `plus` rather than conventional operator symbols, for obvious reasons.

Dummy variables enclosed in double quotes allow an alternative to the pure combinatoric variable-free style of function specification. For example, we could write

```
expectation "t" = times(imean,exp times(mu,"t"))
```

or

```
expectation = "t". times(imean,exp times(mu,"t"))
```

as alternatives to the form shown in Listing 1.3, where the former follows traditional mathematical convention and the latter is more along the lines of "lambda abstraction" familiar to functional programmers.

Use of dummy variables generalizes to higher order functions, for which it is well suited, as seen in the case of the `rho` function. It may also be mixed freely with the combinatoric style. Hence we can write

```
rho "t" =  0.?=/0.! "x". div(...)
```

which says in effect "if the argument to the function returned by `rho` at `"t"` is zero, let that function return a constant value of zero, but otherwise let it return the value of the following expression with the argument substituted for `"x"`."

Abstract data types adhere to a straightforward record-like syntax consisting of a symbolic name for the type followed by square brackets enclosing a comma separated sequence of assignments of values to field identifiers. The values can be of any type, including functions and other records. The `visualization`, `axis`, and `curve` types are used to good effect in this example.

A record is used as an argument to the rendering function because it is useful for it to have many adjustable parameters, but also useful for the parameters to have convenient default settings to spare the user specifying them needlessly. For example, the numbering of the horizontal axes in Listing 1.3 was not explicitly specified but determined automatically by the library, whereas that of the vertical ρ axis was chosen by the user (in the `hatches` field). Values for unspecified fields can be determined by any computable function at run time in a manner inviting comparison with object orientation. Enlightened development with record types is all about designing them with intelligent defaults.

Planar plots

The three dimensional rendering is helpful for intuition but not always a complete picture of the data, and rarely enables quantitative judgements about it. In this example, the dispersion of the peak with increasing time is very clear, but its drift toward higher values of the estimate is less so. A two dimensional plot can be a preferable alternative for some purposes.

Having done most of the work already, we can use the same `visualization` data structure to specify a family of curves in a two dimensional plot. It will not be necessary to recompile the source code for the mathematical model because the data structure storing the samples has been written to a file in binary form.

Listing 1.4 reuse of the data generated by Listing 1.3 for an interpolated 2-dimensional plot

```
#import std
#import nat
#import flo
#import fit
#import lin
#import plo

#output dot'tex' plot

smooth =

~&H\spread visualization$i[
   margin: 15.!,
   picture_frame: ((400.,250.),-30.,-35.)!,
   curves: ~curves; * curve$i[
      points: ^H(*+ ^/~&+ chord_fit0,ari300+ ~&hzXbl)+ ~points,
      attributes: {'linewidth': '0.1pt'}!]]
```

Listing 1.4 shows the required code. Although it would be possible to use the original `spread` record with no modifications, three small adjustments to it are made. These are the kinds of settings that are usually chosen automatically but are nevertheless available to a user preferring more control.

- manual changes to the bounding box (a perennial issue for LaTeX images with no standard way of automatically determining it, the default is only approximate)
- a thinner than default line width for the curves, helpful when many curves are plotted together
- smoothing of the curves by a simple piecewise polynomial interpolation method

Assuming the code in Listing 1.4 is in a file named `smooth.fun`, it is compiled by the command

```
$ fun flo fit lin plo spread smooth.fun
fun: writing `smooth.tex'
```

The command line parameter `spread` is the binary file generated on the previous run. Any binary file included on the command line during compilation is available within the source as a predeclared identifier.

The smoothing effect is visible in Figure 1.4, showing how the resulting plot would appear with smoothing and without. Whereas discernible facets in a three dimensional rendering are a helpful visual cue, line segments in a two dimensional plot are a distraction and should be removed.

A library providing a variety of interpolation methods is distributed with the compiler, including sinusoidal, higher order polynomial, multidimensional, and arbitrary precision

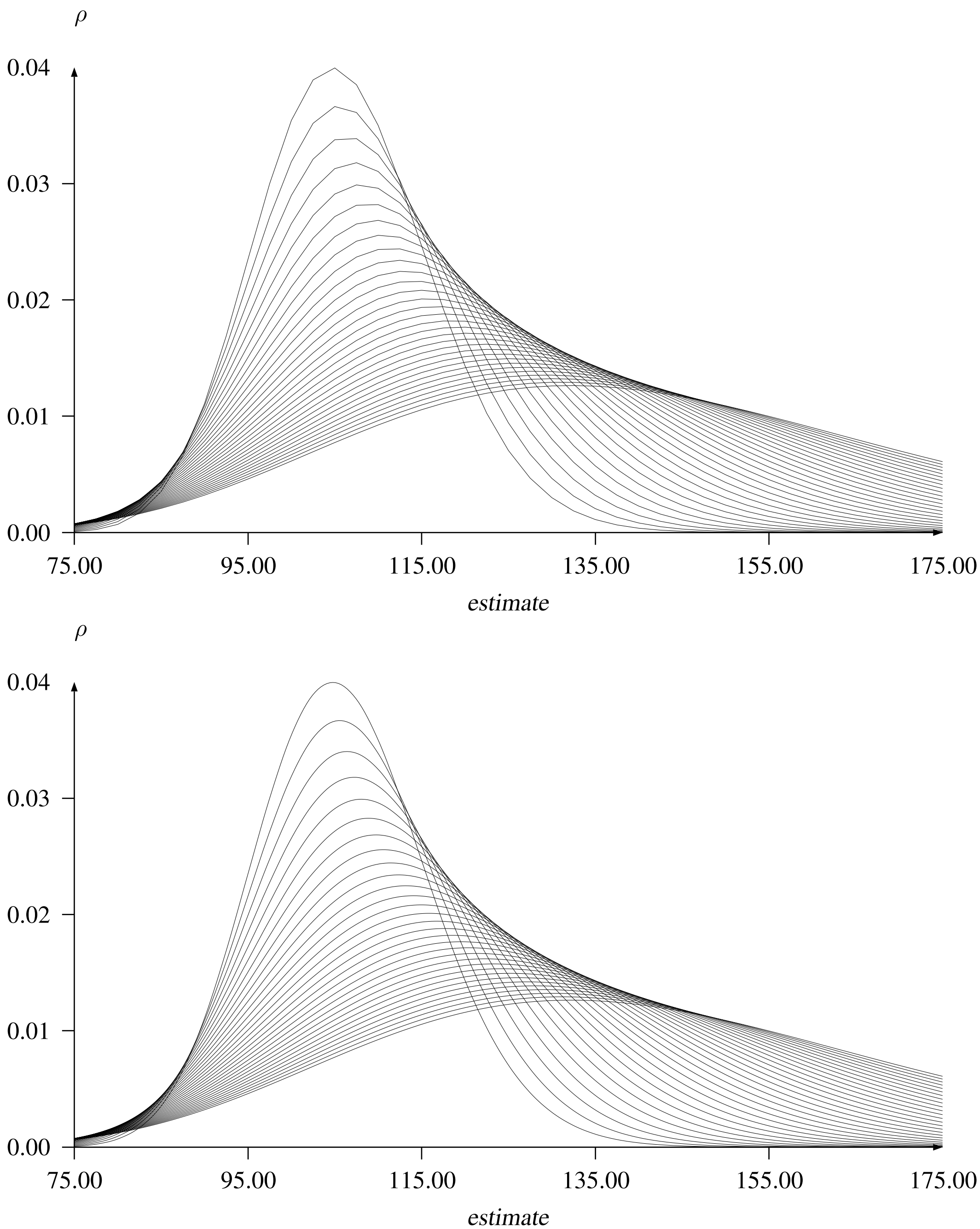


Figure 1.4: plots of data as in Figure 1.3 showing the effects of smoothing

versions. For this example, a simple cubic interpolation (`chord_fit 0`) resampled at 300 points suffices.

1.2.3 Number crunching

For this example, we consider a classic problem in mathematical finance, the valuation of contingent claims (a stuffy name for an interesting problem comparable to finite element analysis). The solution demonstrates some distinctive features of the language pertaining to abstract data types, numerical methods, and GNU Scientific Library functions.

Theory

Two traders want to make a bet on a stock. One of them makes a commitment to pay an amount determined by its future price and the other pays a fee up front. The fee is subject to negotation, and the future payoff can be any stipulated function of the price at that time.

Avoidance of arbitrage One could imagine an enterprising trader structuring a portfolio of bets with different payoffs in different circumstances such that he or she can't lose. So much the better for such a trader of course, but not so for the counterparties who have therefore negotiated erroneous fees.

To avoid falling into this trap, a method of arriving at mutually consistent prices for an ensemble of contracts is to derive them from a common source. A probability distribution for the future stock price is postulated or inferred from the market, and the value of any contingent claim on it is given by its expected payoff with respect to the distribution. The value is also discounted by the prevailing interest rate to the extent that its settlement is postponed.

Early exercise If the claim is payable only on one specific future date, its present value follows immediately from its discounted expectation, but a complication arises when there is a range of possible exercise dates.[4] In this case, a time varying sequence of related distributions is needed.

Binomial lattices A standard construction has a geometric progression of possible stock prices at each of a discrete set of time steps ranging from the contract's inception to its expiration. The sequences acquire more alternatives with the passage of time, and the condition is arbitrarily imposed that the price can change only to one of two neighboring prices in the course of a single time step, as shown in Figure 1.5.

The successor to any price represents either an increase by a factor u or a decrease by a factor d, with $ud = 1$. A probability given by a binomial distribution is assigned to each price, a probability p is associated with an upward movement, and q with a downward movement.

[4] A further complication that we don't consider in this example is a payoff with unrestricted functional dependence on both present and previous prices of the stock.

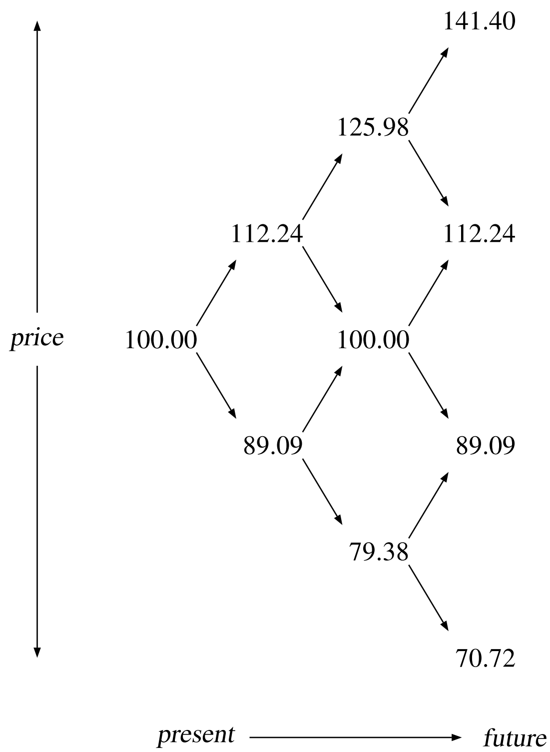


Figure 1.5: when stock prices take a random walk

An astute argument and some high school algebra establish values for these parameters based on a few freely chosen constants, namely Δt, the time elapsed during each step, r, the interest rate, S the initial stock price, and σ, the so called volatility. The parameter values are

$$\begin{aligned} u &= e^{\sigma\sqrt{\Delta t}} \\ d &= e^{-\sigma\sqrt{\Delta t}} \\ p &= \frac{e^{r\Delta t} - d}{u - d} \\ q &= 1 - p \end{aligned}$$

With n time steps numbered from 0 to $n-1$, and $k+1$ possible stock prices at step number k numbered from 0 to k, the fair price of the contract (in this simplified world view) is v_0^0 from the recurrence that associates the following value of v_i^k with the contract at time k in state i.

$$v_i^k = \begin{cases} f(S_i^k) & \text{if} \quad k = n-1 \\ \max\left(f(S_i^k), e^{-r\Delta t}\left(pv_{i+1}^{k+1} + qv_i^{k+1}\right)\right) & \text{otherwise} \end{cases} \tag{1.1}$$

In this formula, f is the stipulated payoff function, and $S_i^k = Su^i d^{k-i}$ is the stock price at time k in state i. The intuition underlying this formula is that the value of the contract at expiration is its payoff, and the value at any time prior to expiration is the greater of its immediate or its expected payoff.

Problem statement

The construction of Figure 1.5, known as a binomial lattice in financial jargon, can be used to price different contingent claims on the same stock simply by altering the payoff function f accordingly, so it is natural to consider the following tasks.

Implement a reusable binomial lattice pricing library allowing arbitrary payoff functions, and an application program for a specific family of functions.

The payoff functions in question are those of the form

$$f(s) = \max(0, s - K)$$

for a constant K and a stock price s. The application should allow the user to specify the particular choice of payoff function by giving the value of K.

Data structures

A lattice can be seen as a rooted graph with nodes organized by levels, such that edges occur only between consecutive levels. Its connection topology is therefore more general than a tree but less general than an unrestricted graph.

An unusual feature of the language is a built in type constructor for lattices with arbitrary branching patterns and base types. Lattices in the language should be understood as containers comparable to lists and sets. For this example, a binomial lattice of floating point numbers is used. The lattice appears as one field in a record whose other fields are the model parameters mentioned above such as the time step durations and transition probabilities.

As indicated above, some of the model parameters are freely chosen and the rest are determined by them. It will be appropriate to design the record data structure in the same way, in that it automatically initializes the remaining fields when the independent ones are given. For this purpose, Listing 1.5 uses a record declaration of the form

$\langle$*record mnemonic*$\rangle$ **: :**
    $\langle$*field identifier*$\rangle$ $\langle$*type expression*$\rangle$ $\langle$*initializing function*$\rangle$
    $\vdots$
    $\langle$*field identifier*$\rangle$ $\langle$*type expression*$\rangle$ $\langle$*initializing function*$\rangle$

If no values are specified even for the independent fields, the record will initialize itself to the small pedagogical example depicted in Figure 1.5.

By way of a demonstration, the code is Listing 1.5 is compiled by the command

```
$ fun flo lat crt.fun
fun: writing `crt.avm'
```

Listing 1.5 implementation of a binomial lattice for financial derivatives valuation

```
#import std
#import nat
#import flo
#import lat

#library+

crr ::

s    %eZ    ~s||100.!
v    %eZ    ~v||0.2!
t    %eZ    ~t||1.!
n    %n     ~n||4!
r    %eZ    ~r||0.05!
dt   %e     ||~dt ~t&& div^/~t float+ predecessor+ ~n
up   %e     ||~up ~v&& exp+ times^/~v sqrt+ ~dt
dn   %eZ    ~v&& exp+ negative+ times^/~v sqrt+ ~dt
p    %eZ    -&~r,~dn,div^(minus^\~dn exp+ times+ ~/r dt,minus+ ~/up dn)&-
q    %eZ    -&~p,fleq\1.+ ~p,minus/1.+ ~p&-
l    %eG

~n&& ~q&& ~l|| grid^(
   ~&lihBZPFrSPStx+ num*+ ^lrNCNCH\~s ^H/rep+~n :^\~&+ ~&h;+ :^^(
      ~&h;+ //times+ ~dn,
      ^lrNCT/~&+ ~&z;+ //times+ ~up),
   ^DlS(
      fleq\;eps++ abs*++ minus*++ div;+ \/-*+ <.~up,~dn>,
      ~&t+ iota+ ~n))

amer = # price of an american option on lattice c with payoff f

("c","f"). ~&H\~l"c" lfold max^|/"f" ||ninf! ~&i&& -+
   \/div exp times/~r"c" ~dt "c",
   iprod/<~q "c",~p "c">+-

euro = # price of a european option on lattice c with payoff f

("c","f"). ~&H\~l"c" lfold ||-+"f",~&l+- ~&r; ~&i&& -+
   \/div exp times/~r"c" ~dt "c",
   iprod/<~q "c",~p "c">+-
```

assuming it resides in a file named `crt.fun`. To see the concrete representation of the default binomial lattice, we display one with no user defined fields as follows.

```
$ fun crt --main="crr&" --cast _crr
crr[
   s: 1.000000e+02,
   v: 2.000000e-01,
   t: 1.000000e+00,
   n: 4,
   r: 5.000000e-02,
   dt: 3.333333e-01,
   up: 1.122401e+00,
   dn: 8.909473e-01,
   p: 5.437766e-01,
   q: 4.562234e-01,
   l: <
      [0:0: 1.000000e+02^: <1:0,1:1>],
      [
         1:1: 1.122401e+02^: <2:1,2:2>,
         1:0: 8.909473e+01^: <2:0,2:1>],
      [
         2:2: 1.259784e+02^: <2:2,2:3>,
         2:1: 1.000000e+02^: <2:1,2:2>,
         2:0: 7.937870e+01^: <2:0,2:1>],
      [
         2:3: 1.413982e+02^: <>,
         2:2: 1.122401e+02^: <>,
         2:1: 8.909473e+01^: <>,
         2:0: 7.072224e+01^: <>]>]
```

In this command, `_crr` is the implicitly declared type expression for the record whose mnemonic is `crr`. The lattice is associated with the field `l`, and is displayed as a list of levels starting from the root with each level enclosed in square brackets. Nodes are uniquely identified within each level by an address of the form $n : m$, and the list of addresses of each node's descendents in the next level is shown at its right. The floating point numbers are the same as those in Figure 1.5, shown here in exponential notation.

Algorithms

Two pricing functions are exported by the library, one corresponding to Equation 1.1, and the other based on the simpler recurrence

$$v_i^k = \begin{cases} f(S_i^k) & \text{if} \quad k = n-1 \\ e^{-r\Delta t}\left(pv_{i+1}^{k+1} + qv_i^{k+1}\right) & \text{otherwise} \end{cases}$$

which applies to contracts that are exercisable only at expiration. The latter are known as European as opposed to American options. Both of these functions take a pair of operands (c, f), whose left side c is record describing the lattice model and whose right side f is a payoff function.

A quick test of one of the pricing functions is afforded by the following command.

```
$ fun flo crt --main="amer(crr&,max/0.+ minus\100.)" --cast
1.104387e+01
```

The payoff function used in this case would be expressed as $f(s) = \max(0, s - 100)$ in conventional notation, and the lattice model is the default example already seen.

As shown in Listing 1.5, the programs computing these functions take a particularly elegant form avoiding explicit use of subscripts or indices. Instead, they are expressed in terms of the `lfold` combinator, which is part of a collection of functional combining forms for operating on lattices defined in the `lat` library distributed with the compiler. The `lfold` combinator is an adaptation of the standard `fold` combinator familiar to functional programmers, and corresponds to what is called "backward induction" in the mathematical finance literature.

The application program

Having made short work of the library, we'll take the opportunity to under-promise and over-deliver by making the application program compute not only the contract prices but also their partial derivatives with respect to the model parameters. These are often a matter of interest to traders, as they represent the sensitivity of a position to market variables.

The source code shown in Listing 1.6 can be used to generate the desired executable program when stored in a file named `call.fun`.

```
$ fun flo crt cop call.fun --archive
fun: writing `call'
```

The `--archive` command line option to the compiler is recommended for larger programs and libraries, and causes the compiler to perform some data compression. In this case it reduces the executable file size by a factor of five, conferring a slight advantage in speed and memory usage. Recall that `crt` is the name of the user written library containing the binomial lattice functions, while `flo` and `cop` are standard libraries distributed with the compiler.

As an executable program, it should be somewhat robust and self explanatory in the handling of input, even if it is used only by its author. When invoked with missing parameters, it responds as follows.

```
$ call
usage: call [-parameter value]* [--greeks]
   -s <initial stock price>
   -t <time to expiration>
   -v <volatility>
```

Listing 1.6 executable program to compute contract prices and partial derivatives

```
#import std
#import nat
#import flo
#import crt
#import cop

usage = # displayed on errors and in the executable shell script

:/'usage: call [-parameter value]* [--greeks]' ~&t -[
   -s <initial stock price>
   -t <time to expiration>
   -v <volatility>
   -r <interest rate>
   -k <strike price>]-

#optimize+

price = # takes a list of parameters to a call option price

<"s","t","v","r","k">. levin_limit amer* *- (
   crr$[s: "s"!,t: "t"!,v: "v"!,r: "r"!,n: ~&]* ~&NiC|\ 8!* iota4,
   max/0.+ minus\"k")

greeks = # takes the same input to a list of partial derivatives

^|T(~&,printf/':%10.3f')*+ -+
   //~&p <'delta','theta','vega ','rho  ','dc/dk','gamma'>,
   ^lrNCT(
      ~&h+ jacobian(1,5) ~&iNC+ price,
      ("h","t"). (derivative derivative price\"t") "h")+-

#comment usage--<'','last modified: '--__source_time_stamp>
#executable (<'par'>,<>)

call = # interprets command line parameters and options

~&iNC+ file$[contents: ~&]+ -+
   ^CNNCT/-+printf/'price:%10.2f',price+~&r+- ~&l&& greeks+ ~&r,
   ~command.options; ^/(any ~keyword[='greeks') -+
      -&~&itZBg,eql/16,all ~&jZ\'0123456789.-'+ ~&h&-?/%ep* usage!%,
      ~parameters*+ ~&itZBFL+ gang *~* ~keyword==* ~&iNCS 'stvrk'+-+-
```

Listing 1.7 executable shell script from Listing 1.6, showing usage and version information

```
#!/bin/sh
# usage: call [-parameter value]* [--greeks]
#    -s <initial stock price>
#    -t <time to expiration>
#    -v <volatility>
#    -r <interest rate>
#    -k <strike price>
#
# last modified: Tue Jan 23 16:14:13 2007
#
# self-extracting with granularity 194
#\
exec avram --par "$0"  "$@"
sSr{EIoAJGhuMsttsp^wZekhsnopfozIfxHoOZ@iGjvwIyd?WwwHoyYnPjo...
...txZEMtpZiKaMS]Mca@ZSC@PUp=O@<
```

```
  -r <interest rate>
  -k <strike price>
```

This message serves as a reminder of the correct way of invoking it, for example

```
$ call -s 100 -t 1 -v .2 -r .05 -k 100
price:     10.45
```

if only the price is required, or

```
$ call -s 100 -t 1 -v .2 -r .05 -k 100 --greeks
price:     10.45
delta:     0.637
theta:     6.412
vega :    37.503
rho  :    53.252
dc/dk:    -0.532
gamma:  1141.803
```

to compute both the price and the "Greeks", or partial derivatives, so called because they are customarily denoted by Greek letters.[5]

Several interesting features of the language are illustrated in this example.

Executable files are requested by the `#executable` compiler directive, and are written as shell scripts that invoke the virtual machine emulator, `avram`, which is not normally visible to the user. The executable files contain a header with some automatically generated front matter and optional comments, as shown in Listing 1.7.

[5]Real users would expect a negative value of Θ, because the value of the contract decays with time. However, the price here has been differentiated with respect to the variable t representing time remaining to expiration, which varies inversely with calendar time.

Command line parsing and validation are chores we try to minimize. One way for an executable program to be specified is by a function mapping a data structure containing the command line options (already parsed) and input files to a list of output files. The command processing in this example program is confined to the last three lines, which verify that each of the five parameters is given exactly once as a decimal number. This segment also detects the `--greeks` flag or any prefix thereof.

Series extrapolation is provided by the `levin_limit` function, which uses the Levin-u transform routines in the GNU Scientific Library to estimate the limit of a convergent series given the first few terms. The convergence of the binomial lattice method is improved in this example by evaluating it for 8, 16, 32, and 64 time steps and extrapolating.

Numerical differentiation is also provided by the GNU Scientific Library, with the help of a couple of wrapper functions. The `derivative` function operates on any real valued function of a real variable, and can be nested to obtain higher derivatives. The `jacobian` function, from the `cop` library distributed with the compiler, takes a pair $(n, m) \in \mathtt{N} \times \mathtt{N}$ to a function that takes a function $f : \mathtt{R}^m \rightarrow \mathtt{R}^n$ to the function $J : \mathtt{R}^m \rightarrow \mathtt{R}^{n \times m}$ returning the Jacobian matrix of the transformation f. The `jacobian` function is convenient for tabulating all partial derivatives of a function of many variables, and adds value to the GSL, whose differentiation routines apply only to single valued functions of a single variable.[6]

1.2.4 Recursive structures

The example in this section demonstrates complex arithmetic, hierarchical data structures, recursion, and tabular data presentation using analogue AC circuit analysis as a vehicle. These are a very simple class of circuits for which the following crash course should bring anyone up to speed.

Theory

Wires in an electrical circuit carry current in a manner analogous to water through a pipe. By convention, a current is denoted by the letter I, and depicted in a circuit diagram by an arrow next to the wire through which it flows.

The rate of current flow is measured in units of amperes. A conservation principle requires the total number of amperes of current flowing into any part of a circuit to equal the number flowing out.

Series combinations This conservation principle allows us to infer that each component of the circuit depicted in Figure 1.6 experiences the same rate of current flow through it, because all are connected end to end. The circle represents a device that propels a fixed

[6]It doesn't take any deliberate contrivance to bump into an undecidable type checking problem. The "type" of the `jacobian` function is $(\mathtt{N} \times \mathtt{N}) \rightarrow ((\mathtt{R}^m \rightarrow \mathtt{R}^n) \rightarrow (\mathtt{R}^m \rightarrow \mathtt{R}^{n \times m}))$ for the particular values of n and m given by the argument to the function, which needn't be stated explicitly at compile time.

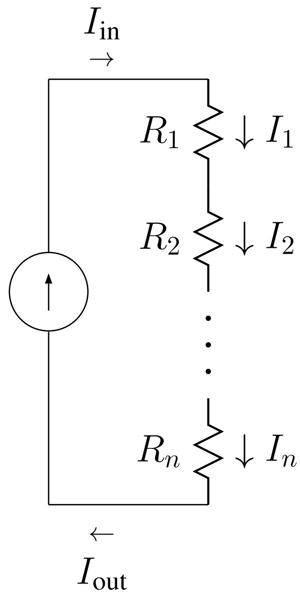


Figure 1.6: resistors in series necessarily carry identical currents, $I_{\text{in}} = I_{\text{out}} = I_k$ for all k

rate of current through itself (a current source), and the zigzagging schematic symbols represent devices that oppose the flow of current through them (resistors).

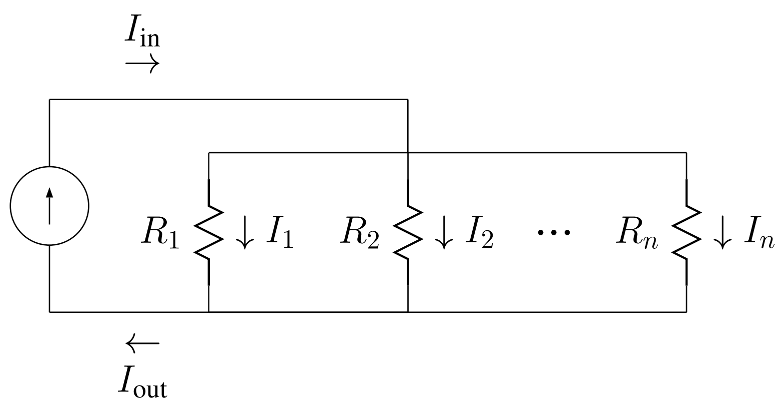


Figure 1.7: rules of current division, $I_{\text{in}} = I_{\text{out}} = \sum I_k$, such that $R_k I_k$ is the same for all k

Parallel combinations A more interesting situation is shown in Figure 1.7, where there are multiple paths for the current to take. In such a case, some fraction of the total current will flow simultaneously through each path. If the resistors along some paths are more effective than others at opposing the flow of current, smaller fractions of the total will flow through them. The effectiveness of a resistor is quantified by a real number R, known as its resistance, expressed in units of ohms (Ω). The current through each path is inversely proportional to its total resistance.

Aggregate resistance It is a consequence of this rule of current division that the effective resistance of a pair of resistors connected in parallel as in Figure 1.7 is the product of their resistances divided by their sum (i.e., $R_1 R_2/(R_1 + R_2)$, for individual resistances R_1 and R_2). Although not directly implied, it is also a fact that the effective resistance of a pair of resistors connected in series as in Figure 1.6 is the sum of their individual resistances.

Normally in a circuit analysis problem the component values are known and the current remains to be determined. The foregoing principles suffice to determine a unique solution for a circuit such as the one shown in Figure 1.8, where the current source emits a current of 10 amperes.

Reactive components For circuits containing only a single fixed current source and resistors connected only in series and parallel combinations, it is easy to imagine a recursive algorithm to determine the current in each branch. Before doing so, we can make matters a bit more interesting by admitting two other kinds of components, an inductor and a capacitor, as shown in Figure 1.9, and allowing the current source to vary with time.

For these components, it is necessary to distinguish between their transient and steady state operation. An inductor will not allow the current through it to change discontinuously. Initially it will prohibit any current at all but gradually will come to behave as a short circuit (i.e., a wire with no resistance). A capacitor behaves in a complementary way, allowing current to flow unimpeded at first but gradually mounting greater opposition until the current direction is reversed.

Individual inductors and capacitors differ in the rate at which they approach their steady state operation in a manner parameterized by a real number L or C, known as their inductance or capacitance, respectively. Without going into detail about the mathematics, suffice it to say that analysis of RLC circuits with time varying sources is of a different order of difficulty than purely resistive networks, requiring in general the solution of a system of simultaneous differential equations.

Complex arithmetic Electrical engineers use an ingenious mathematical shortcut to solve an important special case of RLC circuits algebraically by complex arithmetic without differential equations. A sinusoidally varying current source as a function of time t with constant amplitude I_0, frequency ω and phase ϕ

$$I(t) = I_0 \cos(\omega t + \phi)$$

is identified with a constant complex current

$$I_0 \cos(\phi) + j I_0 \sin(\phi)$$

where the symbol j represents $\sqrt{-1}$.

A generalization of resistance to a complex quantity known as impedance accommodates reactive components as easily as resistors.

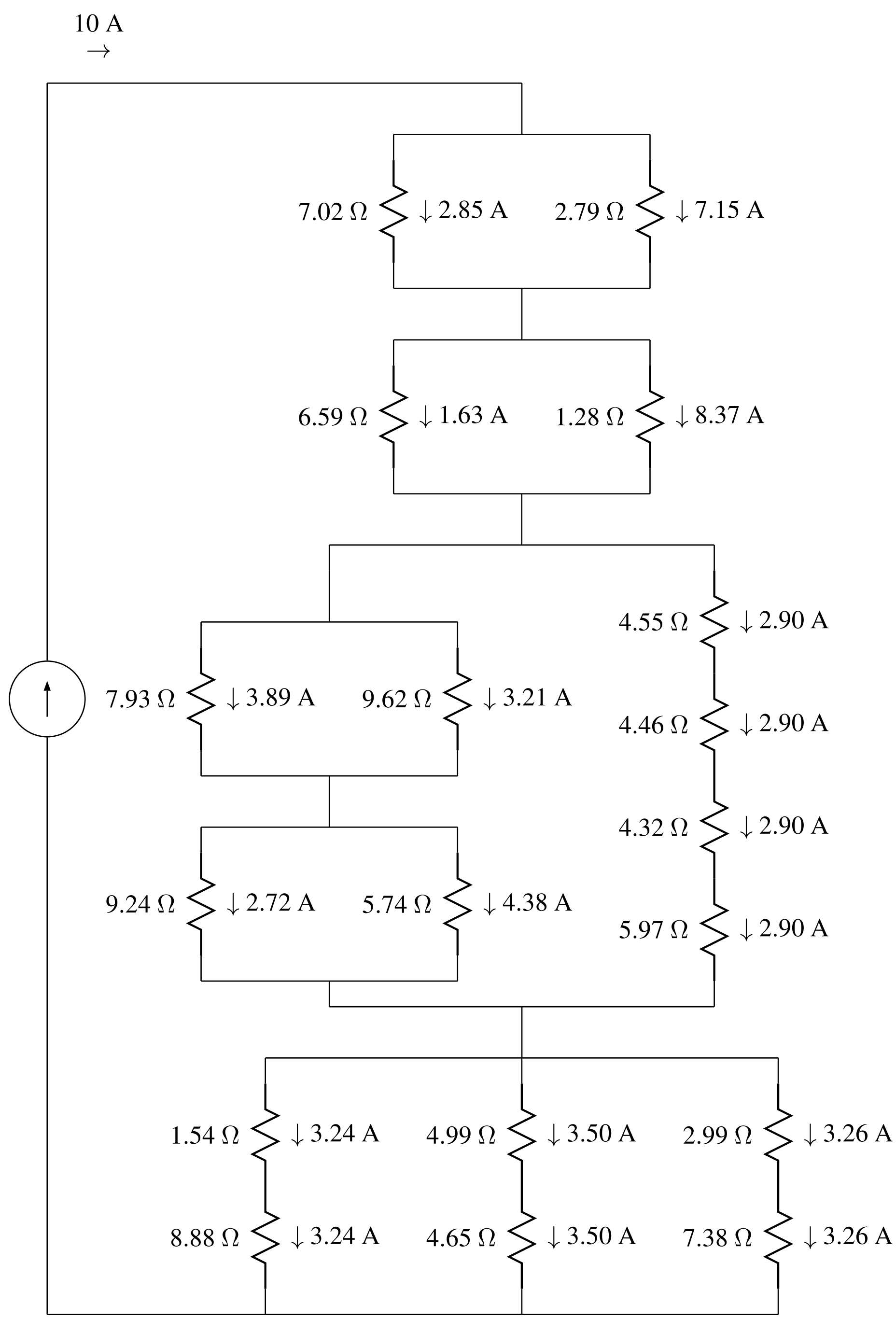


Figure 1.8: any given resistor network implies a unique current division

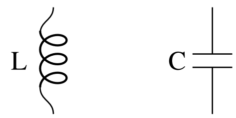


Figure 1.9: An inductor, left, gradually allows current to flow more easily, and a capacitor, right, gradually makes it more difficult

- A resistor with a resistance R has an impedance of $R + 0j$.
- An inductor with an inductance L has an impedance of $j\omega L$, where ω is the angular frequency of the source.
- A capacitor with a capacitance C has an impedance of $-\frac{j}{\omega C}$.

The rules of current division and aggregate impedance for series and parallel combinations take the same form as those of resistance mentioned above, e.g., $Z_1 Z_2/(Z_1 + Z_2)$ for individual impedances Z_1 and Z_2, but are computed by the operations of complex arithmetic. In this way, complex currents are obtained for any branch in a circuit, from which the real, time varying current is easily recovered by extracting the amplitude and phase.

Problem statement

We now have everything we need to know in order to implement an algorithm to solve the following problem.

Exhaustively analyze an AC circuit containing a current source and any series or parallel combination of resistors, capacitors, and inductors.

It is assumed that all component values are known, and the source is sinusoidal with constant frequency, phase, and amplitude. The analysis should be given in the form of a table listing the current and voltage drop across each component in phase and amplitude. The voltage drop follows immediately as the complex product of the current with the impedance.

Data structures

An appropriate data structure for an RLC circuit made from series and parallel combinations is a tree. A versatile form of trees is supported by the language, wherein each node may have arbitrarily many descendents. A tree may have all nodes of the same type, or the terminal nodes can be of a distinct type from the non-terminal nodes.

In this application, each terminal node represents a component in the circuit, and each non-terminal node is a letter, either `` `s `` or `` `p `` for series or parallel combination, respectively. The single back quote indicates a literal character constant in the language.

The components are represented by pairs with a string on the left and a floating point number on the right. The string begins with `R`, `L`, or `C` followed by a unique numerical identifier, and the floating point number is its resistance, inductance, or capacitance, respectively.

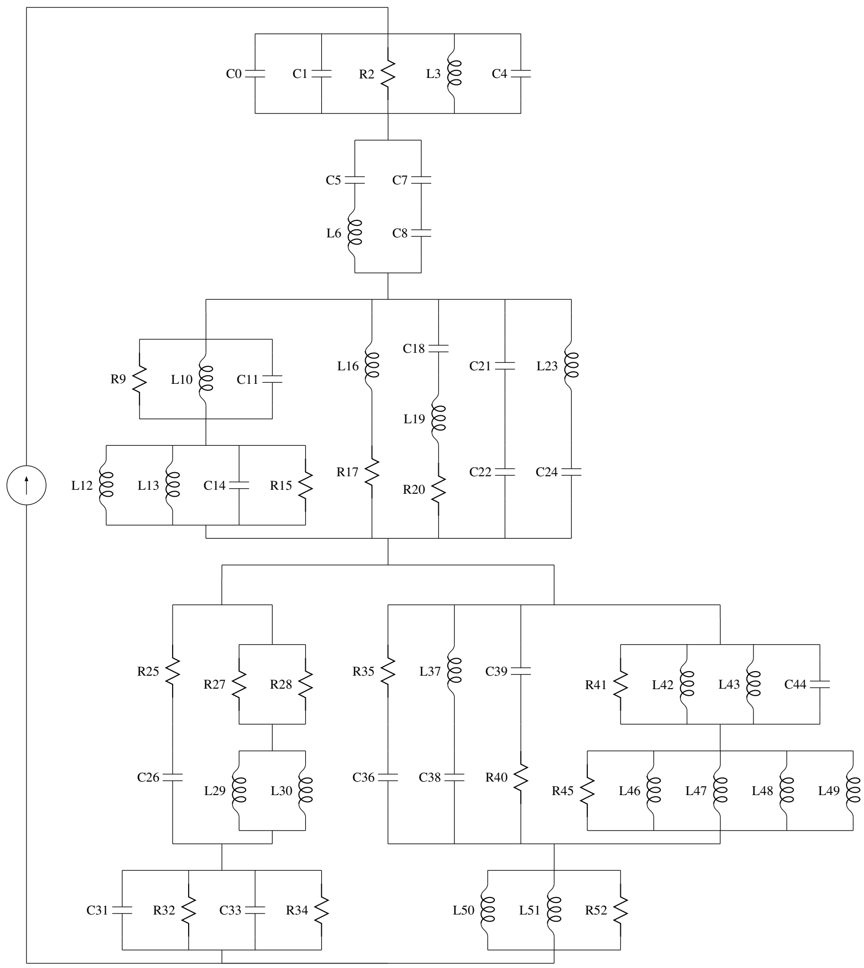


Figure 1.10: an RLC circuit made from series and parallel combinations

The notation for trees used in the language is

$$\langle\mathit{root}\rangle\,\hat{}\,\mathtt{:}\,\mathtt{<[}\langle\mathit{subtree}\rangle\mathtt{[,}\langle\mathit{subtree}\rangle\mathtt{]*]>}$$

where the `^:` operator joins the root to a list of subtrees, each of a similar form, in a comma separated sequence enclosed by angle brackets.

A nice complicated test case for the application is shown in Listing 1.8, which represents the circuit shown in Figure 1.10. This particular example has been randomly generated, but could have been written by hand into a text file. In a real application, the circuit description would probably come from some other program such as a schematic editor.

Following a similar procedure to a previous example, the test data are compiled into a binary file as follows.

```
$ fun circ.fun --binary
fun: writing `circ'
```

Listing 1.8 concrete representation of the circuit in Figure 1.10

```
circ = `s^: <
   `p^: <
      ('C0',5.314278e+00)^: <>,
      ('C1',5.198102e+00)^: <>,
      ('R2',2.552675e+00)^: <>,
      ('L3',3.908299e+00)^: <>,
      ('C4',8.573411e+00)^: <>>,
   `p^: <
      `s^: <('C5',6.398909e+00)^: <>,('L6',1.991548e-01)^: <>>,
      `s^: <('C7',4.471445e+00)^: <>,('C8',4.122309e+00)^: <>>>,
   `p^: <
      `s^: <
         `p^: <
            ('R9',4.076886e+00)^: <>,
            ('L10',4.919520e+00)^: <>,
            ('C11',8.950421e+00)^: <>>,
         `p^: <
            ('L12',2.409632e+00)^: <>,
            ('L13',2.348442e+00)^: <>,
            ('C14',9.192674e+00)^: <>,
            ('R15',3.864372e+00)^: <>>>,
      `s^: <('L16',9.290080e+00)^: <>,('R17',6.017938e+00)^: <>>,
      `s^: <
         ('C18',5.737489e+00)^: <>,
         ('L19',7.591762e+00)^: <>,
         ('R20',8.251754e+00)^: <>>,
      `s^: <('C21',2.025546e+00)^: <>,('C22',4.457961e+00)^: <>>,
      `s^: <('L23',8.891783e+00)^: <>,('C24',7.943625e+00)^: <>>>,
   `p^: <
      `s^: <
         `p^: <
            `s^: <('R25',7.977469e+00)^: <>,('C26',1.069105e+00)^: <>>,
            `s^: <
               `p^: <('R27',8.190201e+00)^: <>,('R28',8.613024e+00)^: <>>,
               `p^: <('L29',9.090409e+00)^: <>,('L30',1.726259e+00)^: <>>>>,
         `p^: <
            ('C31',2.183700e+00)^: <>,
            ('R32',4.809035e+00)^: <>,
            ('C33',1.741527e+00)^: <>,
            ('R34',1.199544e+00)^: <>>>,
      `s^: <
         `p^: <
            `s^: <('R35',6.127510e+00)^: <>,('C36',7.496868e+00)^: <>>,
            `s^: <('L37',4.631129e+00)^: <>,('C38',1.287879e+00)^: <>>,
            `s^: <('C39',2.842224e-01)^: <>,('R40',7.653173e+00)^: <>>,
            `s^: <
               `p^: <
                  ('R41',6.034300e-01)^: <>,
                  ('L42',7.883596e-01)^: <>,
                  ('L43',2.381994e+00)^: <>,
                  ('C44',3.412634e+00)^: <>>,
               `p^: <
                  ('R45',9.246853e+00)^: <>,
                  ('L46',3.435816e+00)^: <>,
                  ('L47',8.543310e+00)^: <>,
                  ('L48',1.537862e+00)^: <>,
                  ('L49',3.412010e+00)^: <>>>>,
         `p^: <
            ('L50',2.899790e+00)^: <>,
            ('L51',7.088897e+00)^: <>,
            ('R52',2.879279e+00)^: <>>>>>
```

Listing 1.9 RLC circuit analysis library using complex arithmetic

```
#import std
#import nat
#import flo

#library+

impedance = # takes a circuit and returns a tree

%cjXsjXDMk+ %ecseXDXCR ~&arv^?(
   ~&ard2falrvPDPMV; ^V\~&v ^/~&d `s?=d(
      ~&vdrPS; c..add:-0,
      ~&vdrPS; :-0 c..div^/c..mul c..add),
   ^:0+ ^/~&ardh case~&ardlh\0! {
      `R: c..add/0+0j+ ~&ardr,
      `L: c..mul/0+1j+ times+~&alrdr2X,
      `C: c..mul/0-1j+ div/1.+ times+~&alrdr2X})

current_division("i","w") = # takes a circuit to a list

%jWmMk+ impedance/"w"; ~&/"i"; ~&arv^?(
   `s?=ardl/~&falrvPDPML ^ML/~&f ^p\~&arv c..mul^*D/~&al -+
      c..vid^*D\~& c..add:-0,
      ~&arvdrPS; c..div/*1.+-,
   ^ANC/~&ardl ^/~&al c..mul+ ~&alrdr2X)

phaser = # returns magnitude and phase in degrees of a complex number

^/..cabs times/180.+ div\pi+ ..carg
```

It is possible to verify that the circuit has been compiled correctly by displaying the binary file contents as a tree type.

```
$ fun circ --main=circ --cast %cseXD
`s^: <
   `p^: <
      ('C0',5.314278e+00)^: <>,
         ...
            ('R52',2.879279e+00)^: <>>>>>
```

The output is seen to match Listing 1.8.

Algorithms

Analysis of the circuit takes place in two passes, the first traversing the tree to determine the aggregate impedance of each subtree, and the second to compute the current division. A separate function for each is defined in Listing 1.9.

The impedance calculation uses a straightforward case statement for terminal nodes corresponding to the bullet point list on page 39. Working from the bottom up, it then performs a cumulative complex summation or parallel combination on these results. Cumulative operations on lists are accomplished without explicit loops or recursion by the reduction combinator, denoted `:-`.

The current division calculation proceeds from the top down, feeding the total input current from above to all subtrees in the case of a series combination, or fractionally for parallel combinations. The precise method used in the latter case is to allocate an input current of

$$\frac{1/Z_k}{\sum 1/Z_n} I_{\text{in}}$$

to the k-th subtree, where I_{in} is the given input current, and Z_k is the impedance of the k-th subtree calculated on the first pass.

Demonstration

To compile the code in Listing 1.9, we first invoke

```
$ fun flo rlc.fun --archive
fun: writing `rlc.avm'
```

The impedance function can be tested with an arbitrarily chosen angular frequency of 1 radian per second and the previously prepared test data file, `circ`.

```
$ fun rlc circ --main="impedance(1.,circ)" --cast %cjXsjXD
(`s,1.143e+00+5.550e-01j)^: <
        ...
              ('R52',2.879e+00+0.000e+00j)^: <>>>>>
```

Here it can be seen that complex numbers are a primitive type defined in the language, with the type mnemonic `j`. The type expression `%cjXsjXD` describes trees whose non-terminal nodes are pairs with characters on the left and complex numbers on the right, and whose terminal nodes are pairs with strings on the left and complex numbers on the right. Although complex numbers are displayed by default with only four digits of precision, the full IEEE double precision format is used in calculations, and other ways of displaying them are possible.

To test the current division function, we choose an input current of $1+0j$ and an angular frequency of 1 radian per second.

```
$ fun rlc circ --m="current_division(1+0j,1.) circ" -c %jWm
<
   'C0': (
      2.821e-01+5.869e-03j,
      1.104e-03-5.308e-02j),
```

⋮

```
   'R52': (
      3.036e-01+2.086e-01j,
      8.741e-01+6.007e-01j)>
```

The result shows the current and voltage drop associated with each component in the circuit, as a pair of complex numbers. The result is given in the form of a list rather than a tree.

Anonymous recursion

The usual way of expressing a recursively defined function in most languages is by writing a specification in which the function is given a name and calls itself. Factorials and Fibonacci functions are the standard examples, which are unnecessary to reproduce here. The compiler is equipped to solve systems of recurrences over functions or other semantic domains in this way, but where functions are concerned, some notational economy is preferable. A noteworthy point of programming style illustrated by the code in Listing 1.9 is the use of anonymous recursion.

A proficient user of the language will find it convenient to express recursive functions in terms of a small selection of relevant combinators such as the recursive conditional denoted `^?`, as shown in Listing 1.9.

Although a list reversal function is available already as a primitive operation, we can express one using this combinator and test it at the same time as follows.

```
$ fun --main="~&a^?(~&fatPRahPNCT,~&a) 'abc'" --cast %s
'cba'
```

Without digressing at this stage for a more thorough explanation, an expanded view of the same program obtained by decompilation gives some indication of the underlying structure of the algorithm.

```
$ fun --m="~&a^?(~&fatPRahPNCT,~&a)" --decompile
main = refer conditional(
   field(0,&),
   compose(
      cat,
      couple(
         recur((&,0),(0,(0,&))),
         couple(field(0,(&,0)),constant 0))),
   field(0,&))
```

On the virtual machine code level, a function of the form `refer f` applied to an argument `x` is evaluated as `f(f,x)`, so that the function is able to access its own machine code as the left side of its operand, and in effect call itself if necessary. Although unconventional, this arrangement is well supported by other language features, and turns out to be the most natural and straightforward approach.

Virtual machine library functions

The complex arithmetic functions such as `c..add` and `c..div` are an example of the general syntax for accessing external libraries linked to the virtual machine, which is

$$\langle \textit{library-name} \rangle \texttt{..} \langle \textit{function-name} \rangle$$

Any library function linked into the virtual machine can be invoked in this way. Both the library name and the function name may be recognizably truncated or omitted if no ambiguity results.

The selection of available library functions is site specific, because it depends on how the virtual machine is configured and on other free software that is distributed separately. An easy way to ascertain the configuration on a given host is to invoke the command

```
$ fun --help library

library functions
------- ---------

:
```

which might display an output similar to Listing 1.10 on a well equipped platform.

Documentation about virtual machine library functions, including their semantics and calling conventions, is maintained with the virtual machine distribution, `avram`, and contained in a reference manual provided in html, info, and postscript formats.

Local additions, modifications or enhancements to virtual machine libraries can be made by a competent C programmer by following well documented procedures, and will be immediately accessible within the language with no modification or rebuilding of the compiler required.

Tabular data presentation

To complete our brief, we need a listing of the amplitude and phase of the voltage and current for each component in tabular form. These data are trivial to extract from a complex number by the hitherto unused function `phaser` defined in Listing 1.9.

```
$ fun rlc --m="phaser 1+1.7320508j" --c %eW
(2.000000e+00,6.000000e+01)
```

The result is a pair of real numbers with the amplitude on the left and the phase in degrees on the right.

Typesetting the table in a manner suitable for publication or presentation eventually will require writing some unpleasant LaTeX code.[7] It would be better for it to be done automatically while the work is ongoing than manually the night before a deadline. To this end, the compiler ships with a library for generating LaTeX tables from a less tedious form of specification.

[7] I'm a big fan of LaTeX because of the quality of the results, but there's no denying that it takes work to get it right.

Listing 1.10 virtual machine libraries displayed by the command `$ fun --help library`

```
library functions
------- ---------
bes     I Isc J K Ksc Y isc j ksc lnKnu y zJ0 zJ1 zJnu
complex add bus cabs cacosh carg casinh catanh ccos ccosh cexp cimag clog conj
        cpow creal create csin csinh csqrt ctan ctanh div mul sub vid
fftw    b_bw_dft b_dht b_fw_dft u_bw_dft u_dht u_fw_dft
glpk    interior simplex
gsldif  backward central forward t_backward t_central t_forward
gslevu  accel utrunc
gslint  qagp qagp_tol qagx qagx_tol qng qng_tol
kinsol  cd_bicgs cd_dense cd_gmres cd_tfqmr cj_bicgs cj_dense cj_gmres cj_tfqmr
        ud_bicgs ud_dense ud_gmres ud_tfqmr uj_bicgs uj_dense uj_gmres uj_tfqmr
lapack  dgeevx dgelsd dgesdd dgesvx dggglm dgglse dpptrf dspev dsyevr zgeevx
        zgelsd zgesdd zgesvx zggglm zgglse zheevr zhpev zpptrf
lpsolve stdform
math    acos acosh add asin asinh asprintf atan atan2 atanh bus cbrt cos cosh
        div exp expm1 fabs hypot isinfinite islessequal isnan isnormal
        isubnormal iszero log log1p mul pow remainder sin sinh sqrt strtod sub
        tan tanh vid
minpack hybrd hybrj lmder lmdif lmstr
mpfr    abs acos acosh add asin asinh atan atan2 atanh bus cbrt ceil
        const_catalan const_log2 cos cosh dbl2mp div div_2ui eint eq equal_p
        erf erfc exp exp10 exp2 expm1 floor frac gamma greater_p greaterequal_p
        grow hypot inf inf_p integer_p less_p lessequal_p lessgreater_p lngamma
        log log10 log1p log2 max min mp2dbl mp2str mul mul_2ui nan nan_p nat2mp
        neg nextabove nextbelow ninf number_p pi pow pow_ui prec root round
        shrink sin sin_cos sinh sqr sqrt str2mp sub tan tanh trunc unequal_abs
        urandomb vid zero_p
mtwist  bern u_cont u_disc u_enum u_path w_disc w_enum
rmath   bessel_i bessel_j bessel_k bessel_y beta dchisq dexp digamma dlnorm
        dnchisq dnorm dpois dt dunif gammafn lbeta lgammafn pchisq pentagamma
        pexp plnorm pnchisq pnorm ppois pt punif qchisq qexp qlnorm qnchisq
        qnorm qpois qt qunif rchisq rexp rlnorm rnchisq rnorm rpois rt runif
        tetragamma trigamma
umf     di_a_col di_a_trp di_t_col di_t_trp zi_a_col zi_a_trp zi_c_col zi_c_trp
        zi_t_col zi_t_trp
```

Listing 1.11 demonstration of circuit analysis and tabular data presentation

```
#import std
#import nat
#import flo
#import rlc
#import tbl

(# quick throwaway program to make a table of voltages and currents
through all components of an RLC circuit read from a binary file
named circ at compile time #)

#binary+

freqs   = <0.1,1.>
data    = ~&hnSPmSSK7p (gang current_division* 1+0j-* freqs) circ
title   = 'componentwise analysis at two frequencies'
content = format/freqs data

#binary-

format = # takes frequencies and data to headings and columns

^|(
   :/<''>^:0+ * -+
      \/~&V ^:(~&iNCNVS <'amplitude','phase'>)* ~&iNCS <
         'current (mA)',
         'voltage drop (mV)'>,
      ~&iNC+ '$\omega = '--+ --'$ rad/s'+ printf/'%0.1f'+-,
   :^/~&nS ~&mS; ~&K7+ *=* --+ phaser;$ ^|lrNCC\~& times/1.e3)

#output dot'tex' label'can'+ elongation title

can = table2 content
```

The `tbl` library is geared toward generating tables with hierarchical headings and columns of numerical or alphabetic data. As Listing 1.11 implies, most of the LaTeX code generation is done by the `table` function, which takes a natural number as an argument specifying the number of decimal places (in this case 2), and returns a function taking a data structure describing the table contents. A couple of other functions deal with the practicalities of the `longtable` format, needed for tables that are too long to fit on a page.

The application in Listing 1.11 is based on the assumption that generating the table will be a one off operation for a particular circuit, rather than justifying the development of a reusable executable as in a previous example. Although not strictly necessary, some of the intermediate data are saved to binary files during compilation for ease of exposition. Compiling the application therefore has the following effect.

```
$ fun flo tbl rlc circ fcan.fun
fun: writing `freqs'
fun: writing `data'
fun: writing `title'
fun: writing `content'
fun: writing `can.tex'
```

The main points to note are that `data` is computed by performing current division over the list of frequencies specified in `freqs`, and transformed to a list of assignments of strings to lists of pairs of complex numbers, as a quick inspection shows.

```
$ fun data --m=data --c %jWLm
<
   'C0': <
      (
         -5.997e-01+3.614e-01j,
         6.800e-01+1.128e+00j),
      (
         2.821e-01+5.869e-03j,
         1.104e-03-5.308e-02j)>,
⋮
   'R52': <
      (
         1.086e-02+7.109e-02j,
         3.125e-02+2.047e-01j),
      (
         3.036e-01+2.086e-01j,
         8.741e-01+6.007e-01j)>>
```

The `content`, in the standard form required by the `table` function, contains a pair whose left side is a list of trees of lists of strings, and whose right side is a list of either lists of strings or lists of floating point numbers.

```
$ fun content --m=content --c %sLTLsLeLULX
(
   <
      <''>^: <>,
      <'$\omega = 0.1$ rad/s'>^: <
         ^: (
            <'current (mA)'>,
            <<'amplitude'>^: <>,<'phase'>^: <>>),
         ^: (
            <'voltage drop (mV)'>,
            <<'amplitude'>^: <>,<'phase'>^: <>>)>,
      <'$\omega = 1.0$ rad/s'>^: <
         ^: (
            <'current (mA)'>,
            <<'amplitude'>^: <>,<'phase'>^: <>>),
         ^: (
            <'voltage drop (mV)'>,
            <<'amplitude'>^: <>,<'phase'>^: <>>)>>,
   <
      <
         'C0',
⋮
         3.449765e+01,
         3.449765e+01>>)
```

Although the trees representing the table headings could have been written out manually, a proficient user will prefer the style shown in Listing 1.11 where possible because it is both shorter and more general, requiring no modification if the list of frequencies is extended or changed in a subsequent run.

The resulting table is shown below.

Table 1.1: componentwise analysis at two frequencies

	$\omega = 0.1$ rad/s				$\omega = 1.0$ rad/s			
	current (mA)		voltage drop (mV)		current (mA)		voltage drop (mV)	
	amplitude	phase	amplitude	phase	amplitude	phase	amplitude	phase
C0	700.18	148.93	1317.54	58.93	282.16	1.19	53.10	-88.81
C1	684.87	148.93	1317.54	58.93	276.00	1.19	53.10	-88.81
R2	516.14	58.93	1317.54	58.93	20.80	-88.81	53.10	-88.81
L3	3371.13	-31.07	1317.54	58.93	13.59	-178.81	53.10	-88.81
C4	1129.58	148.93	1317.54	58.93	455.21	1.19	53.10	-88.81
C5	751.36	0.00	1174.20	-90.00	1101.28	0.00	172.10	-90.00
L6	751.36	0.00	14.96	90.00	1101.28	0.00	219.33	90.00

Table 1.1: componentwise analysis at two frequencies (continued)

	$\omega = 0.1$ rad/s				$\omega = 1.0$ rad/s			
	current (mA)		voltage drop (mV)		current (mA)		voltage drop (mV)	
	amplitude	phase	amplitude	phase	amplitude	phase	amplitude	phase
C7	248.64	0.00	556.07	-90.00	101.28	-180.00	22.65	90.00
C8	248.64	0.00	603.16	-90.00	101.28	-180.00	24.57	90.00
R9	111.87	-77.02	456.08	-77.02	22.10	-87.52	90.11	-87.52
L10	927.09	-167.02	456.08	-77.02	18.32	-177.52	90.11	-87.52
C11	408.21	12.98	456.08	-77.02	806.56	2.48	90.11	-87.52
L12	293.97	-156.84	70.84	-66.84	39.16	-177.35	94.37	-87.35
L13	301.63	-156.84	70.84	-66.84	40.18	-177.35	94.37	-87.35
C14	65.12	23.16	70.84	-66.84	867.52	2.65	94.37	-87.35
R15	18.33	-66.84	70.84	-66.84	24.42	-87.35	94.37	-87.35
L16	86.37	-84.44	80.24	5.56	16.67	-144.50	154.84	-54.50
R17	86.37	-84.44	519.79	-84.44	16.67	-144.50	100.30	-144.50
C18	63.29	-68.86	110.31	-158.86	16.63	-129.39	2.90	140.61
L19	63.29	-68.86	48.05	21.14	16.63	-129.39	126.23	-39.39
R20	63.29	-68.86	522.25	-68.86	16.63	-129.39	137.20	-129.39
C21	73.25	14.34	361.63	-75.66	256.94	2.56	126.85	-87.44
C22	73.25	14.34	164.31	-75.66	256.94	2.56	57.64	-87.44
L23	1422.67	14.34	1265.00	104.34	21.05	-177.44	187.13	-87.44
C24	1422.67	14.34	1790.95	-75.66	21.05	-177.44	2.65	92.56
R25	22.28	132.96	177.73	132.96	167.17	44.75	1333.58	44.75
C26	22.28	132.96	208.39	42.96	167.17	44.75	156.36	-45.25
R27	33.42	81.44	273.73	81.44	154.95	19.00	1269.07	19.00
R28	31.78	81.44	273.73	81.44	147.34	19.00	1269.07	19.00
L29	10.41	81.44	9.46	171.44	48.24	19.00	438.56	109.00
L30	54.80	81.44	9.46	171.44	254.05	19.00	438.56	109.00
C31	15.88	163.23	72.74	73.23	246.62	42.97	112.94	-47.03
R32	15.13	73.23	72.74	73.23	23.48	-47.03	112.94	-47.03
C33	12.67	163.23	72.74	73.23	196.68	42.97	112.94	-47.03
R34	60.64	73.23	72.74	73.23	94.15	-47.03	112.94	-47.03
R35	22.11	93.52	135.49	93.52	48.54	30.31	297.44	30.31
C36	22.11	93.52	29.49	3.52	48.54	30.31	6.48	-59.69
L37	18.99	171.24	8.79	-98.76	77.18	-60.94	357.44	29.06
C38	18.99	171.24	147.46	81.24	77.18	-60.94	59.93	-150.94
C39	3.85	158.97	135.50	68.97	35.32	53.75	124.27	-36.25
R40	3.85	158.97	29.47	158.97	35.32	53.75	270.32	53.75
R41	103.15	78.34	62.24	78.34	370.47	-68.29	223.55	-68.29
L42	789.54	-11.66	62.24	78.34	283.57	-158.29	223.55	-68.29
L43	261.31	-11.66	62.24	78.34	93.85	-158.29	223.55	-68.29
C44	21.24	168.34	62.24	78.34	762.91	21.71	223.55	-68.29
R45	8.28	83.60	76.56	83.60	42.65	63.27	394.35	63.27
L46	222.84	-6.40	76.56	83.60	114.78	-26.73	394.35	63.27
L47	89.62	-6.40	76.56	83.60	46.16	-26.73	394.35	63.27
L48	497.87	-6.40	76.56	83.60	256.43	-26.73	394.35	63.27
L49	224.40	-6.40	76.56	83.60	115.58	-26.73	394.35	63.27
L50	714.06	-8.68	207.06	81.32	365.74	-55.50	1060.58	34.50

Table 1.1: componentwise analysis at two frequencies (continued)

	$\omega = 0.1$ rad/s				$\omega = 1.0$ rad/s			
	current (mA)		voltage drop (mV)		current (mA)		voltage drop (mV)	
	amplitude	phase	amplitude	phase	amplitude	phase	amplitude	phase
L51	292.09	-8.68	207.06	81.32	149.61	-55.50	1060.58	34.50
R52	71.91	81.32	207.06	81.32	368.35	34.50	1060.58	34.50

1.3 Remarks

Not every capability of the language has been illustrated in this chapter, but at this point most readers should have a pretty good idea about whether they want to know more. In any case, grateful acknowledgement is due to all those who have graciously read this far with an open mind. The assumption henceforth is that readers who are still reading have made a commitment to learn the language, so that less space needs to be devoted to motivation.

1.3.1 Installation

The compiler is distributed in a `.tar` archive or a git repository available from

`http://www.gueststar.github.com/Ursala`

In order for it to work, it depends on the `avram` virtual machine emulator, available from

`http://www.gueststar.github.com/Avram`

Please refer to the `avram` documentation for installation instructions.

Some optional external libraries usable by `avram` are recommended but not required, notably the `mpfr` library for arbitrary precision arithmetic. Arbitrary precision floating point numbers are normally a primitive type in the language, but are disabled without this library.[8]

Nomenclature

Since its earliest prototypes, the name of the compiler has been `fun`, and this name is retained because of its brevity and the ease typing it on a command line. However, the transformation from personal tool kit to a community project necessitates a more recognizable and searchable name in the interest of visibility. The name of Ursala has been chosen for the language as of this release, which is meant as a quasi-abbreviation for "universal applicative language". This manual uses the word Ursala to refer to the language in the abstract (*e.g.*, "a program written in Ursala") and `fun` in typewriter font to refer to the compiler.

[8]Arbitrary precision natural and rational numbers and fixed precision floating point numbers are available regardless.

Root installations

The compiler may be installed either system-wide or for an individual user. For the former case, the system administrator (i.e., the `root` user) needs to place the executable and library files under apporpriate standard directories. The system administrator should unpack the `.tar` archive and copy the files as shown.

```
$ tar -zxf ursala-0.1.0.tar.gz
$ cp ursala-0.1.0/bin/* /usr/local/bin
$ mkdir /usr/local/lib/avm
$ chmod ugo+rx /usr/local/lib/avm
$ cp ursala-0.1.0/src/*.avm /usr/local/lib/avm
$ cp ursala-0.1.0/lib/*.avm /usr/local/lib/avm
```

Use of these standard directories is advantageous because it will allow the virtual machine to locate the library files automatically without requiring the user to specify their full paths.

Non-root installations

If the compiler is installed only for an individual user, the libraries and executables should be unpacked as above, but can be moved to whatever directories the user prefers and can access. The virtual machine will not automatically detect libraries in non-standard directories, but on a GNU/Linux system it can be made to do so by way of the `AVMINPUTS` environment variable. For example, if the user wishes to store a collection of personal library modules under `$HOME/avm`, the command

```
$ export AVMINPUTS=".:$HOME/avm"
```

either executed interactively or in a `bash` initialization script will enable it. The syntax for equivalent commands may differ with other shells.

Porting

There is no provision for installation on other operating systems (for example Microsoft Windows), but volunteer efforts in that connection are welcome. Other solutions (short of free software advocacy in general) such as emulation or use of the Cygnus tools are also an option but are beyond the scope of this document.

Virtual machine code applications are entirely portable to any platform on which the virtual machine is installed, subject only to the requirement that any optional virtual machine modules used by the application are also installed on the target platform. Even this modest requirement can be flexible if the developer makes use of run-time detection features and replacement functions.

1.3.2 Organization of this manual

Anyone wishing to use Ursala effectively should read Part II on language elements and Part III on standard libraries, whereas only those wishing to modify or enhance the compiler itself should read Part IV on compiler internals. Because the language is much more extensible than most, the latter group should also read the rest of the manual first to establish that the enhancements they require are not more easily obtained by less heroic means. Part III assumes a working knowledge of Part II, and Part IV assumes a guru-level knowledge of Parts II and III.

The chapters in Part II are meant to be read sequentially on a first reading, with each covering a particular topic about the language. Although one may argue for a more intuitive order of presentation, this need must be balanced against that of maintainability of the document itself, in anticipation of possible contributions by other authors over the life of the project. If any chapter in Part II becomes particularly rough going on a first reading, the reader is invited to jump to the concluding remarks of that chapter for a summary and proceed to the next one.

A convention is followed whereby minimal amounts material may be introduced out of turn where necessary for continuity if they are useful for an explanation of a topic at hand, but are nevertheless fully documented in their appropriate chapter even if some repetition occurs.

Whereas the main text can be read sequentially, certain code fragments designated as example programs may depend on material not yet introduced at the point where they are listed. These can be skipped on a first reading without loss of continuity. It is considered more important to demonstrate optimal use of all relevant language features at all times than to insist on continuity in the examples.

1.3.3 License

The compiler and this documentation are Copyright 2007-2012 by Dennis Furey. This document is freely distributed under the terms of the GNU Free Documentation License, version 1.2, with no front cover texts, no back cover texts, and no invariant sections. A copy of this license is included in Appendix B.

The compiler and supporting modules are distributed according to Version 3 of the General Public License as published by the Free Software Foundation. Anyone is allowed to copy, modify, and redistribute the software or works derived from it under compatible terms, whether commercially or otherwise, but not to turn it into a closed source product or to encumber it with Digital Restrictions Management directed against the end user. Please refer to the GPL text for full details. If you think you have an ethical justification for distributing it under different terms (e.g., confidentiality of medical records, defiance of oppressive regimes, *etcetera*), contact the author or the current maintainer at `ursala-users@freelists.org`.

Use of the compiler incurs no obligation in itself to distribute anything. Moreover, applications compiled by the compiler are not necessarily derivative works and theoretically could be distributed under a non-free license. However, compiled applications that are distributed under a non-free license must avoid dependence on any functions found in the `.avm` supporting modules distributed with the compiler, such as the standard library `std.avm`, because an effect of compilation would be to copy the library code into them.

End users of applications developed with the compiler will need a virtual machine to execute them. Whether the applications are free or not, there is no legal impediment to using `avram` for this purpose, provided it is distributed according to the terms of its license, the GPL, and provided the license for the application permits disassembly, without which it can't be executed. No individual is able to authorize alternative distribution terms for `avram` because it depends on contributions by many copyright holders.

Part II

Language Elements

So we need machines and they need us. Is that your point, councillor?

Neo in *The Matrix Reloaded*

2
Pointer expressions

Much of the expressive power of the language derives from a concise formalism to encode combinations of frequently used operations. These come under the general name of pointers or pointer expressions, although this term does not adequately convey the versatility of this mechanism, which has no counterpart in other modern languages. This chapter explains everything there is to know about pointer expressions.

2.1 Context

Syntactically a pointer expression is a case sensitive string of letters or digits appearing as a suffix of an operator to qualify its meaning in some way. The concepts of operators, operands, and operator suffixes are developed more fully in Chapters 5 and 6, but in order to discuss pointer expressions, two particularly relevant operators are necessary to introduce in advance.

- The ampersand operator, `&`, with no suffix evaluates to the identity pointer, and with a suffix evaluates to the pointer that the suffix describes.
- The field operator, `~`, is a prefix operator taking a pointer as an operand, and evaluates to the function induced by it.

A distinction is made between a pointer and the function induced by it (e.g., the identity pointer versus the identity function), because it is possible and often useful to manipulate or transform pointers directly in ways that are not applicable to functions. This distinction is also reflected in the underlying virtual machine code representation.

Listing 2.1 the left deconstructor function the hard way

```
#library+

f("x","y") = "x"
```

2.2 Deconstructors

The simplest kinds of functions induced by pointers are known variously as projections, deconstructions, or generalized identity functions, but in this manual the term deconstructors is preferred.

2.2.1 Specification of a deconstructor

A deconstructor is a function that takes some type of aggregate data structure as an argument, and returns some component of its argument as a result.

To illustrate this concept, we can consider the problem of implementing a program to compute the following function.

$$f(x, y) = x$$

That is to say, the function should take a pair of operands, and return the left side.

One way of implementing it in Ursala would be with dummy variables, as shown in Listing 2.1. To see that this implementation is perfectly correct, we compile it as shown,

```
$ fun dum.fun
fun: writing `dum.avm'
```

and now try it out on a few examples.

```
$ fun dum --main="f('foo','bar')" --cast
'foo'
$ fun dum --main="f(123,456)" --cast
123
$ fun dum --main="f()" --cast
fun:command-line: invalid deconstruction
```

Conveniently, the function is naturally polymorphic, and the `--cast` option is smart enough to guess the result type if it's something simple. The function inherently raises an exception if its argument isn't a pair of anything, but luckily the compiler does a reasonable job of exception handling.

2.2.2 Deconstructor semantics

Expressing a deconstructor function in this way amounts to writing an equation for the compiler to solve, and it is instructive to exhibit the solution directly.

```
$ fun dum --main=f --decompile
main = field(&,0)
```

This result shows the virtual machine code for the left deconstructor function, which consists of the `field` combinator, a common feature of all deconstructor functions corresponding to the `~` operator in the language, and the expression `(&,0)`, which represents a pointer to the left.

The notation used to display the pointer in the decompiled code is actually a syntactically sugared form of a type of ordered binary trees with empty tuples for leaves. The zero represents the empty tuple and the ampersand represents a pair of empty tuples, which can be made explicit with an appropriate cast. (More about type casts is explained in Chapter 3.)

```
$ fun --main="(&,0)" --cast %hhZW
(((),()),())
```

Pointer expressions therefore store no information other than that which is embodied in their shape. Their rôle is simply to specify the displacement of a subtree with respect to the root of an ordered binary tree of any type. The pointer referring to the right of a pair would be `(0,&)`, the pointer to the right of the left of a pair of pairs would be `((0,&),0)`, and so on.

2.2.3 Deconstructor syntax

A primary design goal of this language to be as concise as possible. Rather than using nested tuples, equations, or verbose mnemonics, the left and right deconstructor functions can be expressed directly as `~&l` and `~&r`, respectively, using built in pointer expressions. These equivalences can be verified as shown.

```
$ fun --main="&l" --cast %t
(&,0)
$ fun --main="&r" --cast %t
(0,&)
$ fun --m="~&l" --decompile
main = field(&,0)
$ fun --m="~&r" --decompile
main = field(0,&)
$ fun --m="~&l ('foo','bar')" --c
'foo'
```

Nested deconstructors

Further benefits of this syntax accrue in more complicated deconstructions. To get to the left of the right of a pair of pairs, we write `~&lr`, to get to the right of the right or the left of the left, we write `~&rr` or `~&ll`, respectively, and so on to arbitrary depths.

```
$ fun --m="~&ll (('a','b'),('c','d'))" --c
'a'
$ fun --m="~&lr (('a','b'),('c','d'))" --c
'b'
$ fun --m="~&rl (('a','b'),('c','d'))" --c
'c'
$ fun --m="~&rr (('a','b'),('c','d'))" --c
'd'
```

Compound deconstructors

Deconstruction functions can also be made to retrieve more than one field from an argument, by using a tuple of pointers.

```
$ fun --m="~(&lr,&rl) (('a','b'),('c','d'))" --c
('b','c')
$ fun --m="~(&rl,&lr) (('a','b'),('c','d'))" --c
('c','b')
```

Note that the order of the pointers in the tuple determines the order in which the fields are returned.

When a tuple of deconstructors is used, the result type is considered a tuple. To express the notion of a compound deconstructor returning a list, a colon can be used.

```
$ fun --m="~&r:&l (<1,2,3>,0)" --c
<0,1,2,3>
$ fun --m="~&h:&tt <0,1,2,3>" --c
<0,2,3>
```

The pointer on the left side of the colon accounts for the head of the result, and the one on the right accounts for the tail.

The colon has other uses in the language. In pointer expressions, it must be without any adjacent white space to ensure correct disambiguation.

Nested compound deconstructors

A form of relative addressing takes place when a compound deconstructor is nested.

```
$ fun --m="~(0,(&r,&l)) (('a','b'),('c','d'))" --c
('d','c')
```

In this example, the `&l` and `&r` deconstructors refer not to the whole argument but to the part on the right, due to their offset within the pointer where they occur.

A better notation for compound deconstructors is introduced shortly, using constructors. However, the notation shown here is applicable in certain situations where the alternative isn't, namely whenever pointer expressions are designated by user defined identifiers.

			deconstructors			
	constructor		primary		secondary	
type class	operation	mnemonic	operation	mnemonic	operation	mnemonic
pairs	cross	`X`	left	`l`	right	`r`
lists	cons	`C`	head	`h`	tail	`t`
sets	-	-	element	`e`	subset	`u`
assignments	assign	`A`	name	`n`	meaning	`m`
trees	vertex	`V`	root	`d`	subtrees	`v`
jobs	join	`J`	function	`f`	argument	`a`

Table 2.1: pointer expressions for constructors and deconstructors

Miscellaneous deconstructors

A way to get the same field out of both sides of a pair of pairs is to use the `b` deconstructor as follows.

```
$ fun --m="~&bl (('a','b'),('c','d'))" --c
('a','c')
$ fun --m="~&br (('a','b'),('c','d'))" --c
('b','d')
```

The identity deconstructor, `i`, refers to the whole argument, as does an empty pointer expression.

```
$ fun --m="~&i 'me'" --c
'me'
$ fun --m="~& 'myself'" --c
'myself'
```

See Section 2.3.2 for motivation.

2.2.4 Other types of deconstructors

Pairs aren't the only aggregate data type in Ursala. There are also lists, sets, assignments, trees, and jobs. Each has its own operator syntax and its own deconstructors corresponding to `&l` and `&r`, as shown in Table 2.1. The deconstructors are the main concern at present. Here is an example of each.

```
$ fun --main="~&h <'a','b'>" --cast
'a'
$ fun --main="~&t <'a','b'>" --cast
<'b'>
$ fun --main="~&e {'a','b'}" --cast
'a'
$ fun --main="~&u {'a','b'}" --cast %S
```

```
{'b'}
$ fun --main="~&n 'a': 'b'" --cast
'a'
$ fun --main="~&m 'a': 'b'" --cast
'b'
$ fun --main="~&d 'a'^:<'b'^: <>>" --cast
'a'
$ fun --main="~&vh 'a'^:<'b'^: <>>" --cast %T
'b'^: <>
$ fun --main="~&f ~&J('a','b')" --cast
'a'
$ fun --main="~&a ~&J('a','b')" --cast
'b'
```

Note that the subtrees of a tree, referenced by `~&v`, are a list of trees, the head of the list of subtrees, obtained by `~&vh`, is a tree, but `~&vhd` would refer to the root node in the first subtree. This expression mixes tree deconstructors with a list deconstructor, which is perfectly valid. Any types of deconstructors can be mixed in the same expression, with the obvious interpretation.

The concept of different classes of aggregate types is an artifact of the language rather than the virtual machine. On the virtual machine level, all aggregate data types are represented as pairs, all primary deconstructors listed in Table 2.1 have the representation `(&,0)`, and all secondary deconstructors have the representation `(0,&)`. Use of the appropriate deconstructor for a given type is not enforced. For example, `~&r <x,y,z>` could be written in place of `~&t <x,y,z>`, and both would evaluate to `<y,z>`. Needless to say, the latter is preferred because well typed code is easier to maintain unless there is a compelling reason for writing it otherwise, but the language design stops short of insisting on it to the point of overruling the programmer.

2.3 Constructors

The next simplest form of pointer expressions are the constructors, as shown in Table 2.1, namely `X`, `C`, `V`, `A`, and `J`. Each constructor complements a pair of deconstructors, and serves the purpose of putting two fields together into an aggregate type.

2.3.1 Constructors by themselves

One way for these constructors to be used is in functions such as `~&X`, which take a pair of arguments and return the aggregate as a result. Each side of the following expressions is equivalent to the other.

```
 ~&X(x,y)  ≡  (x,y)
~&C(x,<y>) ≡  <x,y>
 ~&V(x,y)  ≡  x^:y
 ~&A(x,y)  ≡  x: y
```

- There is no operator notation in the language for the job constructor, `J`.
- The usage of `~&X` in this way is always superfluous, because its argument is already a pair, so it serves as the identity function of pairs.

Another way for these constructors to be used is with an empty argument, `()`, in which case they designate the empty instance of the relevant type. For example, `~&C()` ≡ `<>`. A notion of empty tuples, trees, assignments, and jobs is implied, but there is no particular notation for the latter three.

2.3.2 Constructors in expressions

The real reason for these constructors to exist is to be used in pointer expressions, which make it easy for data to be taken apart and put together in a different way. A pointer expression containing a constructor has a left subexpression, followed by a right subexpression, followed by the constructor, with no intervening space. The subexpressions can be deconstructors or nested expressions with constructors.

For example, the pointer expression shown below interchanges the sides of a pair.

```
%$
$ fun --main="~&rlX (1.,2.)" --cast
(2.000000e+00,1.000000e+00)
```

This one repeats the first item of a list, using the hitherto unmotivated identity deconstructor, `i`.

```
%$
$ fun --main="~&hiC <'foo','bar'>" --cast
<'foo','foo','bar'>
```

This one takes the head of a list of pairs with its left and right sides interchanged.

```
$ fun --main="~&hrlX <(1,2),(3,4),(5,6)>" --cast
(2,1)
```

2.3.3 Disambiguation issues

In more complicated cases, a minor difficulty arises. If we consider the problem of a pointer expression to delete the second item of a list, we might think to write `&httC`, with the intent that the left subexpression is `h` and the right one is `tt`. However, this idea won't work.

```
$ fun --main="~&httC <0,1,2,3>" --cast
fun:command-line: invalid deconstruction
```

The problem is that the `C` constructor applies only to the two subexpressions immediately preceding it, `tt`, and the `h` is interpreted as the offset for the rest. The result is equivalent to the nested compound deconstruction `(&t:&t,0)`, which attempts to deconstruct the first item of the list (in this case `0`), and additionally attempts to create a badly typed list whose head is the same as its tail. The exception is due to the first issue.

It would be possible to fall back on the usage `&h:&tt` demonstrated on page 59, but this problem justifies a more comprehensive solution without extra punctuation. The `P` constructor can be used in this connection to group two subexpressions into an indivisible unit. The meaning of `ttP` is the same as that of `tt`, but the former is treated as a single subexpression in any context.

Revisiting the example with the correct pointer expression usage, we have

```
$ fun --m="~&httPC <'a','b','c','d','e'>" --c
<'a','c','d','e'>
```

These constructors can be arbitrarily nested.

```
$ fun --m="~&htttPPC <'a','b','c','d','e'>" --c
<'a','d','e'>
```

Because repetitions are frequent, a natural number expressed in decimal can be substituted in any pointer expression for that number of consecutive occurrences of the `P` constructor.

```
$ fun --m="~&httt2C <'a','b','c','d','e'>" --c
<'a','d','e'>
```

2.3.4 Miscellaneous constructors

Two further pointer constructors, `G` and `I` are also defined. Each of these requires two subexpressions, similarly to the constructors discussed above.

Glomming

The simplest way to give a semantics for the `G` constructor is as follows. For any function of the form `~&`uv`X` that returns a result of the form `(a,(b,c))` when applied to an argument x, the function `~&`uv`G` returns the result `((a,b),(a,c))` when applied to the same x. That is, a copy of the left is paired up with each side of the right.

One consequence of this semantics is that `~&lrG` can be written as a shorter form of `~&lrlPXlrrPXX`. If a pointer expression begins with `lrG`, it can be shortened further by omitting the initial `lr` because they are inferred.

expression	equivalent	effect on $((a, b), (c, d))$
`&bbI`	`&llPrlPXlrPrrPXX`	$((a, c), (b, d))$
`&brlXI`	`&lrPrrPXllPrlPXX`	$((b, d), (a, c))$
`&rlXbI`	`&rlPllPXrrPlrPXX`	$((c, a), (d, b))$
`&rlXrlXI`	`&rrPlrPXrlPllPXX`	$((d, b), (c, a))$

Table 2.2: using `I` for rotations and reflections of a pair of pairs

Pairwise relative addressing

The `I` constructor has four practical uses shown in Table 2.2, as well as any generalizations of those obtained by using `lrX` in place of `b` and/or any single valued deconstructor in place of `r` or `l`. Other generalizations can be used experimentally but their effect is unspecified and subject to change in future revisions.

2.4 Pseudo-pointers

The pointer expression syntax is such a convenient way of specifying constructors and deconstructors that it has been extended to more general functions. Pointer expressions describing more general functions are called pseudo-pointers in this manual. The virtual machine code for a pseudo-pointer is not necessarily of the form `field` f. For example,

```
$ fun --main="~&L" --decompile
main = reduce(cat,0)
```

However, pseudo-pointers can be mixed with pointers in the same expression, as if they were ordinary constructors or deconstructors. For example,

```
$ fun --m="~&hL" --d
main = compose(reduce(cat,0),field(&,0))
```

For the most part, it is not necessary to be aware of the underlying virtual machine code representation, unless the application is concerned with program transformation. Most operators in Ursala that allow pointer expressions as suffixes also allow pseudo-pointers. The exception is the `&` operator, which is meaningful only if its suffix is really a pointer.

```
$ fun --main="&L" --cast %t
fun:command-line: misused pseudo-pointer
```

As a matter of convenience, there is an exception to the exception, which is the case of a function of the form `~&`p. Recall that the `~` operator maps a pointer operand to the function induced by it. The semantics of this expression where p is a pseudo-pointer is the function specified by p, even though `&`p would not be meaningful by itself.

	meaning	example		
L	list flattening	~&L <<1>,<2,3>,<4>>	≡	<1,2,3,4>
N	empty constant	~&N x	≡	0
s	list to set conversion	~&s <'c','b','b','a'>	≡	{'a','b','c'}
x	list reversal	~&x <3,6,1>	≡	<1,6,3>
y	lead items of a list	~&y <'a','b','c','d'>	≡	<'a','b','c'>
z	last item of a list	~&z <'a','b','c','d'>	≡	<'d'>

Table 2.3: pseudo-pointers represent more general functions than deconstructors

2.4.1 Nullary pseudo-pointers

Some pseudo-pointers may require subexpressions to precede them in a pointer expression, similarly to constructors such as X and C, while others are analogous to primitive operands like t and r in the algebra of pointer expressions. Examples of the latter are shown in Table 2.3.

Some of these, such as the lead and last items of a list, are obvious complements to operations expressible by pointers, and are defined as pseudo-pointers only because they are inexpressible by the virtual machine's field combinator. Others may seem unrelated to the kinds of transformations lending themselves to pointer expressions, but in fact were chosen as pseudo-pointers precisely because they occur frequently in the same context.

List flattening

The L pseudo-pointer describes the function that converts a list of lists into one long list by forming the cumulative concatenation of the items. This function is also useful on character strings, which are represented as lists of characters.

Empty constant

The N can be used in a pointer wherever it is convenient to have a constant empty value stored in the result. One example would be a usage like ~&NrX which takes a pair of operands (x,y) and returns (0,y), with any value of x replaced by 0. A more frequent usage is in the expression ~&iNC, which forms the cons of the argument with the empty list, thereby returning a unit list <x> for any argument x.

List to set conversion

Sets are represented in the language as lexically ordered lists with no duplicates. The ~&s function takes any list as an argument and returns the set of its items, by sorting them and removing duplicates.

List reversal

The reversal of a list begins with the last item, followed by the second to last, and so on back to the first. A fast, constant space implementation of list reversal at the virtual machine level is accessible by the `~&x` function. List reversal is often needed in practical algorithms.

Lead items of a list

The `~&y` function takes a list as an argument and returns the list obtained by deleting the last item. The length of the result is one less than the length of the original. An exception is thrown if this function is applied to an empty list.

Last item of a list

The `~&z` function takes a list as an argument and returns the last item. This function is implemented by a constant number of virtual machine operations but actually takes a time proportional to the length of the list. An exception is raised in the case of an empty list as an argument.

A small example of rolling a list to the right are as follows.

```
$ fun --m="~&zyC 'abcd'" --c
'dabc'
```

One way of rolling to the left would be by reversal before and after rolling to the right.

```
$ fun --m="~&xzyCx 'abcd'" --c
'bcda'
```

Although each of `x`, `y`, and `z` requires a list reversal when used by itself, the compiler automatically performs global optimizations on pseudo-pointer expressions that sometimes remove unnecessary operations.

```
$ fun --main="~&xzyCx" --decompile
main = compose(
   reverse,
   couple(field(&,0),compose(reverse,field(0,&))))
```

Note that the virtual machine's `reverse` function appears only twice rather than three or four times in the compiled code.

Example program

A small example demonstrating a couple of these operations in context is shown in Listing 2.2. This example uses some language features not yet introduced, and may either be skipped on a first reading of this manual or read with partial comprehension by the following explanation.

Listing 2.2 some pseudo-pointers and a pointer in a practical setting

```
#import std

#comment -[This program reads a text file from standard input and
writes it to standard output with all tab characters replaced by the
string '<tab>'.]-

#executable &

showtabs = * ~&L+ * (~&h skip/9 characters)?=/'<tab>'! ~&iNC
```

Listing 2.3 executable file from Listing 2.2

```
#!/bin/sh
# This program reads a text file from standard input and
# writes it to standard output with all tab characters replaced by the
# string '<tab>'.
#\
exec avram  "$0"  "$@"
uIzMOt[QV]uGmzlSgcr>=d\nT\
```

The application is meant to display text files containing tab characters in such a way that the tabs are explicit, as opposed to being displayed as spaces. It does so by substituting each tab character with the string `<tab>`.

The algorithm applies a function to each character in the file. The function maps the tab character to the `'<tab>'` character string, but maps any other character to the string containing only that character, using `~&iNC`.

When this function is applied to every character in a string, the result is a list of character strings, which is flattened into a character string by `~&L`. This operation is applied to every character string in the file.

One other pointer expression in this example is `&h`, which is used to define a compile-time constant. The tab character is the ninth character (numbered from zero) in the list of characters defined in the standard library, which is computed as the head of the list of characters obtained by skipping the first nine. This computation is performed at compile time and does not require any search of the character table at run time.

To compile the program, we run the command

```
$ fun showtabs.fun
fun: writing `showtabs'
```

This operation generates a free standing executable, as shown in Listing 2.3

A peek at the virtual machine code is easy to arrange for enquiring minds (possibly to the detriment of the obfuscation research community). The executable code stored in binary format can be accessed like any other data file during a subsequent compilation.

```
$ fun showtabs --m=showtabs --decompile
```

combinator usage	interpretation
$\texttt{reduce(}f\texttt{,}k\texttt{)}\ \texttt{<>}$	k
$\texttt{reduce(}f\texttt{,}k\texttt{)}\ \texttt{<}a\texttt{,}b\texttt{,}c\texttt{,}d\texttt{>}$	$f\texttt{(}f\texttt{(}a\texttt{,}b\texttt{)}\texttt{,}f\texttt{(}c\texttt{,}d\texttt{))}$
$\texttt{map(}f\texttt{)}\ \texttt{<}a\ldots z\texttt{>}$	$\texttt{<}f\texttt{(}a\texttt{)}\ldots f\texttt{(}z\texttt{)>}$
$\texttt{conditional(}p\texttt{,}f\texttt{,}g\texttt{)}\ \ x$	if $p\texttt{(}x\texttt{)}$ then $f\texttt{(}x\texttt{)}$ else $g\texttt{(}x\texttt{)}$
$\texttt{compose(}f\texttt{,}g\texttt{)}\ \ x$	$f\texttt{(}g\texttt{(}x\texttt{))}$
$\texttt{constant(}k\texttt{)}\ \ x$	k
$\texttt{compare(}x\texttt{,}y\texttt{)}$	if $x=y$ then `true` else `false`
$\texttt{cat(<}x_0\ldots x_n\texttt{>,<}y_0\ldots y_m\texttt{>)}$	$\texttt{<}x_0\ldots y_m\texttt{>}$
$\texttt{couple(}f\texttt{,}g\texttt{)}\ \ x$	$\texttt{(}f\texttt{(}x\texttt{)}\texttt{,}g\texttt{(}x\texttt{))}$

Table 2.4: informal and incomplete virtual machine quick reference

```
main = map compose(
   reduce(cat,0),
   map conditional(
      compose(
         compare,
         couple(constant <0,&,0,0,0>,field &)),
      constant '<tab>',
      couple(field &,constant 0)))
```

The strange looking constant is the concrete representation of the tab character. An intuitive listing of some other combinators in this code is shown in Table 2.4, but are more formally documented in the `avram` reference manual.

The following small test file will be the input.

```
$ cat /etc/crypttab
# <target name> <source device>         <key file>
cswap   /dev/hda3       /dev/random
```

Most of the spaces shown above are due to tabs. We can now use the compiled program to display the tabs explicitly.

```
$ showtabs < /etc/crypttab
# <target name><tab><source device><tab><tab><key file>
cswap<tab>/dev/hda3<tab>/dev/random
```

The input file, incidentally, is not valid as a real crypttab.

2.4.2 Unary pseudo-pointers

The versatility of pointer expressions is further advanced by a selection of pseudo-pointers representing functional combining forms, shown in Table 2.5. Unlike ordinary pointer constructors, these require only a single subexpression, but the identity pointer, `i`, is inferred as a subexpression if nothing precedes them in the expression. The semantics of

	meaning	example		
F	filter combinator	~&tFL <<1,2>,<3>,<4,5>>	≡	<1,2,4,5>
S	map combinator	~&rlXS <(0,1),(2,3)>	≡	<(1,0),(3,2)>
Z	negation	~&iZS <true,false,true>	≡	<false,true,false>
g	list conjunction	~&lg <(1,'a'),(0,'b')>	≡	0
k	list disjunction	~&rk <('x','y'),('z','')>	≡	true
o	tree folding	~&dvLPCo `a^:<`b^:0,`c^:0>	≡	'abc'

Table 2.5: unary pseudo-pointers provide functional combinators within pointer expressions

most of these pseudo-pointers should be nothing new to functional programmers, but are nevertheless explained in this section.

Logical operations

Some of these pseudo-pointers involve logical operations (i.e., operations pertaining to whether something is true or false). The standard library defines constants `true` and `false`, which are represented respectively as `((),())` and `()`, and can also be written as `&` and `0`.

Most standard functions returning a logical value will return one of the above, but any value of any type can also be identified with a logical value. Empty lists, empty tuples, empty sets, empty strings, empty instances of trees, jobs, or assignments, and the natural number zero are all logically equivalent to `false` in this language. Any non-empty value of any type including functions, characters, real numbers, and type expressions is logically equivalent to `true`.

This convention simplifies the development of user defined predicates by removing the need for explicit conversion to logical values. For example, the predicate to test for non-emptiness of a list is simply the identity function, `~&`. This function obviously will return the whole list, but when it's used as a predicate, returning the whole list is the same as returning `true` if the list is non-empty, and `false` otherwise.

Filter combinator

The `F` pseudo-pointer requires a pointer or function computing a predicate as a subexpression, in the sense described above. The result is a function mapping lists to lists, that works by applying the predicate to every item of the input list and retaining only those items in the output for which the predicate returns a non-empty value.

For example, the function `~&iF` or simply `~&F` removes the empty items from a list. The function shown in Table 2.5 takes a list of lists and removes the items containing only a single item (and hence empty tails). It also flattens the result using `L`.

Map combinator

The map pseudo-pointer, denoted `S`, requires a subexpression operating on the items of a list, and specifies a function that operates on a whole list by applying it to each item and making a list of the results. Maps in functional languages are as commonplace as loops in imperative languages.

Negation

Negation is expressed by the `Z` pseudo-pointer, and has the effect of inverting the logical value returned by the function or pointer in its subexpression. That is, false values are changed to true and true values are changed to false.

List conjunction

The `g` pseudo-pointer expresses list conjunction, which is the operation of applying a predicate to every item of a list and returning a true value if and only if every result is true (with truth understood in the sense described above).

A single false result refutes the predicate and causes the algorithm to terminate without visiting the rest of the list. There is a slight advantage in execution time if it occurs close to the beginning of the list.

List disjunction

A complementary operation to the above, list disjunction, denoted `k`, involves applying a predicate to every item of a list and returning a true result if any of the individual results is true. The list traversal halts when the first true result is obtained.

Relationships among these logical operations follow well known algebraic laws, which the compiler uses to perform code optimization on pointer expressions.

Tree folding

This operation is somewhat more involved than the others. The tree folding pseudo-pointer, denoted `o`, requires a subexpression representing a function that will be used to obtain a result by traversing a tree from the bottom up.

The function described by the subexpression is expected to take a tree as an argument, whose root is the node of the input tree currently being visited, and whose subtrees are the list of results computed previously when the subtrees of the current node were visited. This list will be empty in the case of terminal nodes. The result returned by the function can be of any type.

The function is not required to cope with the case of an empty tree. If the whole argument is an empty tree, then the result is `0` regardless of the function. If the argument is not empty but some subtrees of it are, those will appear as zero values in the list of subtrees passed to the function when their parent node is visited.

The simple example of `~&dvLPCo` shown in Table 2.5 may help to make the matter more concrete. This function will take a tree of anything and make a list of the nodes in the order they would be visited by a preorder traversal.

- The subexpression contains the function `~&dvLPC`.
- This function forms a list as the cons of the results of the two functions `~&d` and `~&vLP`.
- The `~&d` function accesses the root datum of the subtree currently being visited.
- The `~&vL` function takes the list of results previously computed for the subtrees, `~&v`, which will be a list of lists, and flattens them into one list with `L`.
- With the root on the left and the resulting list from the subtrees on the right, the result for whole tree is obtained by the cons operation, `C`.

The example therefore shows that a tree of characters is mapped to a character string.

Correct parsing

Some attention to detail is required to use these pseudo-pointers correctly. Because the subexpression of a unary pseudo-pointer is always required (except in the case of an implied identity deconstructor at the beginning of an expression), there is no need to use the `P` constructor to make them an indivisible unit as described in Section 2.3.3. For example, writing `hFP` instead of `hF` is unnecessary. In fact, it is an error, and worse yet, it might not be flagged during compilation if another subexpression precedes it, which the `P` will then include.

On the other hand, it may well be necessary to group the subexpression of a unary pseudo-pointer using `P`. For example, the expression `hhS` is not equivalent to `hhPS`.

Writing complicated pointer expressions can be error prone even for an experienced user of Ursala. Learning to read the decompiled listings can be a helpful troubleshooting technique.

2.4.3 Ternary pseudo-pointers

There are two ternary pseudo-pointers, denoted by `q` and `Q`. Each of them requires three subexpressions to precede it in the pointer expression. The first subexpression represents a predicate, the second represents a function to be applied if the predicate is true, and the third represents a function to be applied if the predicate is false.

Semantics

The `conditional` combinator in the virtual machine directly supports this operation for both pseudo-pointers, as shown in Table 2.4. The lower case `q` additionally wraps the resulting virtual machine code in the `refer` combinator, which has the property

$$\forall f.\ \forall x.\ (\texttt{refer}\ f)(x) = f(\texttt{~\&J}\ (f, x))$$

That is to say, the f in a function of the form `refer` f accesses the original argument to the outer function `refer` f by `~&a`, and accesses a copy of itself by `~&f`. Recall from Table 2.1 that `~&f` and `~&a` are the deconstructors associated with the job constructor `~&J`.

Non-self-referential conditionals

An example of the `Q` pseudo-pointer is given by the function `~&lNrZQ`, defining a binary predicate that returns a true value if and only if neither of its operands is true.

```
$ fun --m="~&lNrZQS <(0,0),(0,1),(1,0),(1,1)>" --c %bL
<true,false,false,false>
```

The function is shown here mapped over the list of all possible combinations so as to exhibit its truth table. Conditional combinators are used in two places, one for the `Q` and one for the `Z`.

```
$ fun --main="~&lNrZQ" --decompile
main = conditional(
   field(&,0),
   constant 0,
   conditional(field(0,&),constant 0,constant &))
```

Recursion

It is impossible to give a good example of the `q` pseudo-pointer without introducing a binary pseudo-pointer `R`. This pseudo-pointer requires two subexpressions to precede it in the pointer expression where it occurs, unless it is at the beginning of the expression, in which case the subexpressions `lr` are inferred.

The `R` pseudo-pointer occurring in a pointer expression of the form `~&`fa`R` has the following property.

$$\forall f.\ \forall a.\ \forall x.\ \texttt{~\&}fa\texttt{R}\ (x) = (\texttt{~\&}f\ x)\ (\texttt{~\&J}(\texttt{~\&}f\ x, \texttt{~\&}a\ x))$$

This property holds for any pointer expressions f and a, not necessarily identical to the deconstructors `f` and `a`.

The purpose of the `R` pseudo-pointer is to perform a "recursive call" to a function that is given as some part of the argument, by applying it to some other part of the argument. In operational terms, the first subexpression f should manipulate x to produce the virtual machine code for a function to be called, and the second subexpression a should construct or retrieve some component of x to serve as the argument in the recursive call.

When the recursive call is performed, the function obtained by f is applied not just to the argument obtained by a, but to the job containing both the function and the argument. In this way, the function has access to its own machine code and can make further recursive calls if necessary. This mechanism is inherent in the `R` pseudo-pointer.

Self-referential conditionals

As an example of the `q` pseudo-pointer, we can implement the following function that performs a truncating zip operation. The truncating zip of a pair of lists forms the list of pairs obtained by pairing up the corresponding items from the lists. If one list has fewer items than the other, the trailing items on the longer list are ignored. That is, for a pair of lists

$$(\langle x_0, x_1 \dots x_n \rangle, \langle y_0, y_1 \dots y_m \rangle)$$

the result of the truncating zip is the list of pairs

$$\langle (x_0, y_0), (x_1, y_1) \dots (x_k, y_k) \rangle$$

where $k = \min(n, m)$.

The specification for this function is `~&alrNQPabh2fabt2RCNq`, which is first demonstrated and then explained further.

```
$ fun --m="~&alrNQPabh2fabt2RCNq ('ab','cde')" --c
<(`a,`c),(`b,`d)>
```

Recall that character strings enclosed in forward quotes are represented as lists of characters, and that individual character constants are expressed using a back quote.

The virtual machine code for the function is as follows.

```
$ fun --m="~&alrNQPabh2fabt2RCNq" --decompile
main = refer conditional(
   conditional(field(0,(&,0)),field(0,(0,&)),constant 0),
   couple(
      field(0,(((&,0),0),(0,(&,0)))),
      recur((&,0),(0,(((0,&),0),(0,(0,&)))))),
   constant 0)
```

The `recur` combinator in the virtual code directly corresponds to the `R` pseudo-pointer for the important special case of subexpressions that are pointers rather than pseudo-pointers.

- The three main subexpressions are `alrNQP`, `abh2fabt2RC`, and `N`.
- The predicate `alrNQP` tests whether both sides of the argument are non-empty.
- The third subexpression `N` is applied when the predicate doesn't hold (i.e., when at least one side of the argument is empty), and returns an empty list.
- The middle subexpression, `abh2fabt2RC`, is applied when both sides of the argument are non-empty.
  - The `C` pseudo-pointer makes this subexpression return a list whose head is computed by `abh2` and whose tail is computed `fabt2R`
  - The pair of heads of the argument is accessed by `abh2`.
  - A recursive call is performed by `fabt2R`, with the function and the pair of tails.

	meaning	example		
B	conjunction	`~&ihBF <0,1,2,3>`	≡	`<1,3>`
D	left distribution	`~&zyD <0,1,2>`	≡	`<(2,0),(2,1)>`
E	comparison	`~&blrE ((0,1),(1,1))`	≡	`(false,true)`
H	function application	`~&lrH (~&x,'abc')`	≡	`'cba'`
M	mapped recursion	`~&aaNdCPfavPMVNq 1^:<2^:0,3^:0>`	≡	`2^:<4^:0,6^:0>`
O	composition	`~&blrEPlrGO (1,(1,2))`	≡	`(true,false)`
R	recursion	`~&aafatPRCNq 'ab'`	≡	`<'ab','b'>`
T	concatenation	`~&rlT ('abc','def')`	≡	`'defabc'`
U	union of sets	`~&rlU ({'a','b'},{'b','c'})`	≡	`{'a','b','c'}`
W	pairwise recursion	`~&afarlXPWaq ((0,&),(&,&))`	≡	`((&,&),(&,0))`
Y	disjunction	`~&lrYk <(0,0),(0,1),(0,0)>`	≡	`true`
c	intersection of sets	`~&lrc ({'a','b'},{'b','c'})`	≡	`{'b'}`
j	difference of sets	`~&hthPj <{'a','b'},{'b','c'}>`	≡	`{'a'}`
p	zip function	`~&lrp (<1,2>,<3,4>)`	≡	`<(1,3),(2,4)>`
w	membership	``~&nmw `b: 'abc'``	≡	`true`

Table 2.6: binary pseudo-pointers add greater utility to pointer expressions

2.4.4 Binary pseudo-pointers

An assortment of pseudo-pointers taking two subexpressions provides a diversity of useful operations. The two subexpressions should immediately precede the binary pseudo-pointer in a pointer expression, but may be omitted if they are the deconstructors `lr` and are at the beginning of the expression (e.g., `~&p` may be written for `~&lrp`).

The alphabetical list of binary pseudo-pointers is shown in Table 2.6, but they are grouped by related functionality in this section for expository purposes. The areas are list operations, recursion, set operations, logical operations, and general purpose functional combinators.

List operations

To start with the easy ones, there are three frequently used list operations provided by binary pseudo-pointers.

T – concatenation Both subexpressions are expected to return lists when evaluated, and the result from `T` is the list obtained by concatenating the first with the second.

The concatenation of two lists $\langle x_0 \dots x_n\rangle$ and $\langle y_0 \dots y_m\rangle$ is defined as the list

$$\langle x_0 \dots x_n, y_0 \dots y_m\rangle$$

containing the items of both, with the order and multiplicity preserved, and with the items of the left preceding those of the right. More formally, it satisfies these equations.

```
      ~&T(<>,y)  =  y
~&T(~&C(h,t),y)  =  ~&C(h,~&T(t,y))
```

Note that concatenation is not commutative, so `~&rlT` shown in Table 2.6 differs from `~&T`, which is short for `~&lrT`.

D – left distribution The second subexpression of the `D` pseudo-pointer is expected to return a list, and each item of it is paired up with a copy of the result returned by the first subexpression. Each pair has the first subexpression's result on the left and the list item on the right. The complete result is a list of pairs in order of the list returned by the right subexpression.

More formally, the `D` pseudo-pointer is that which satisfies these equations, where the subexpressions `lr` are implicit.

$$\begin{aligned} \texttt{\~{}\&D(}x\texttt{,<>)} &= \texttt{<>} \\ \texttt{\~{}\&D(}x\texttt{,\~{}\&C(}h\texttt{,}t\texttt{))} &= \texttt{\~{}\&C((}x\texttt{,}h\texttt{),\~{}\&D(}x\texttt{,}t\texttt{))} \end{aligned}$$

p – zip function Both subexpressions are expected to return lists of the same length, and the result of the `p` pseudo-pointer is the list of pairs made by pairing up the corresponding items. A specification in a similar style to those above would be as follows.

$$\begin{aligned} \texttt{\~{}\&p(<>,<>)} &= \texttt{<>} \\ \texttt{\~{}\&p(\~{}\&C(}x\texttt{,}t\texttt{),\~{}\&C(}y\texttt{,}u\texttt{))} &= \texttt{\~{}\&C((}x\texttt{,}y\texttt{),\~{}\&p(}t\texttt{,}u\texttt{))} \end{aligned}$$

This function contrasts with the truncating zip function used in a previous example (page 73) by being undefined if the lists are of unequal lengths.

```
$ fun --m="~&p(<1,2,3>,<1,2,3,4>)" --c
fun:command-line: invalid transpose
```

Recursion

Each of the following three pseudo-pointers uses the first subexpression to retrieve the code for a function to be invoked, which must be already inherent in the argument, and the second subexpression to retrieve the data to which it is applied. They differ in calling conventions for the function.

R – recursion The simplest form of recursion pseudo-pointer, `R`, is introduced on page 72 in connection with the recursive conditional pseudo-pointer `q`, but briefly repeated here for completeness.

To evaluate a pointer expression of the form `~&`fa`R` with an argument x, the function `~&`f x retrieved by the first subexpression is applied to the job `~&J(~&`f x`,~&`a x`)`. Both the function and the data are passed to the function so that further invocations of itself are possible.

A simple example of tail recursion as in Table 2.6 is the following.

```
$ fun --m="~&aafatPRCNq 'abcde'" --c
<'abcde','bcde','cde','de','e'>
```

The recursive call, `fatPR` applies the function to the tail of the argument, while the enclosing subexpression `afatPRC` forms the list with the whole argument at the head and the result of the recursive call in the tail. The alternative subexpression `N` returns an empty list in the base case.

`M` – mapped recursion This variation on the recursion pseudo-pointer may be more convenient for trees and other data structures where a function is applied recursively to each of a list of operands. The first subexpression retrieves the function, as above, but the second subexpression retrieves a list of operands rather than just one operand. The mapping of the function over the list is implicit.

To be precise, a pointer expression of the form `~&`fa`M` applied to an argument x will return a list of the form

$$\langle(\texttt{~\&}f\ x)\ (\texttt{~\&J}(\texttt{~\&}f\ x, a_0))\dots(\texttt{~\&}f\ x)\ (\texttt{~\&J}(\texttt{~\&}f\ x, a_n))\rangle$$

where `~&`$a\ x = \langle a_0 \dots a_n\rangle$.

Normally a recursively defined function is written with the assumption that the `~&f` field of its argument is a copy of itself, which this semantics accommodates without the programmer distributing it explicitly over the list. Otherwise, it would be necessary to write `~&`fa`DlrRSP` to achieve the same effect as `~&`fa`M`, with the difficulty escalating in cases of nested recursion or other complications.

The example in Table 2.6 uses this pseudo-pointer to traverse a tree of natural numbers from the top down, returning a tree of the same shape with double the number at each node. It relies on the fact that natural numbers are represented as lists of bits with the least significant bit first, so any non-zero natural number can be doubled by the function `~&NiC`, which inserts another zero bit at the head.

In the expression `aaNdCPfavPMVNq`, the recursive call `favPM` has the function addressed by `f` and the list of subtrees addressed by `avP` as subexpressions to the `M` pseudo-pointer. The double of the root is computed by `aNdCP`, and the resulting tree is formed by the `V` constructor.

`W` – pairwise recursion This pseudo-pointer is similar to the above except that it recursively applies a function to each side of a pair of operands rather than to each item of a list. That is, a pointer expression of the form `~&`fa`W` applied to an argument x will return a pair of the form

$$((\texttt{~\&}f\ x)\ (\texttt{~\&J}(\texttt{~\&}f\ x, a_l)), (\texttt{~\&}f\ x)\ (\texttt{~\&J}(\texttt{~\&}f\ x, a_r)))$$

where `~&`$a\ x = (a_l, a_r)$.

Set operations

As mentioned previously, sets are represented as ordered lists with duplicates removed. Three pseudo-pointers directly manipulate sets in this form. The subexpressions associated with these pseudo-pointers are each expected to return a set.

`U` – union of sets This pseudo-pointer returns the union of a pair of sets, which contains every element that is a member of either or both sets. The result may be incorrect if either operand does not properly represent a set as an ordered list without duplicates. However, any list can be put into this form by the `s` pseudo-pointer, as described on page 65.

`c` – intersection of sets This pseudo-pointer returns the set of elements that are in members of both sets. It will also work on unordered lists and lists containing duplicates.

`j` – difference of sets This pseudo-pointer returns the set of elements that are members of the set obtained from the first subexpression and not members of those obtained from the second. It will also work on unordered lists and lists containing duplicates.

Logical operations

There are four binary logical operations implemented by pseudo-pointers. Logical values are understood in the sense described on page 69. That is, anything empty is false and anything non-empty is true.

`B` – conjunction This pseudo-pointer performs a non-strict conjunction, which is to say that it returns a true value if and only if both of its subexpressions returns a true value, but it doesn't evaluate the second subexpression if the first one is false.

In the case of a false value, `0` is returned, but in the alternative, the value of the second subexpression is returned, as the virtual machine code shows.

```
$ fun --m="~&B" --d
main = conditional(field(&,0),field(0,&),constant 0)
```

An application can take advantage of this semantics, for example, by using `~&ihB` to return the head of a list if the list is non-empty, and a value of zero otherwise. The function `~&ihB` will also test whether a natural number is odd without causing an invalid deconstruction when applied to zero.

`Y` – disjunction This pseudo-pointer performs a non-strict disjunction in a manner analogous to the previous one. That is, it returns a true value if either of its subexpressions returns a true value, but doesn't evaluate the second one if the first one is true.

If the first subexpression is true, its value is returned. Otherwise, the value of the second subexpression is returned.

`E` – comparison This pseudo-pointer compares the results returned by its two subexpressions, both of which are always evaluated, and returns a value of `&` (true) if they are equal or zero otherwise. Unlike the preceding pseudo-pointers, it does not necessarily return the value of a subexpression.

Equality in this context is taken to mean that the two results have the same virtual machine code representation. It is possible for two values of different types to be equal if their representations coincide. It is also possible for two semantically equivalent instances of the same abstract data type to be unequal if their representations differ. Functions can also be compared, and only their concrete representations are considered.

The criteria for equality do not include being stored in the same memory location on the host, this concept being foreign to the virtual code semantics, so any two structurally equivalent copies of each other are equal. However, comparison is supported by a virtual machine instruction whose implementation transparently detects pointer equality (in the conventional sense of the words) and manages shared data structures so that comparison is a fast operation on average.

It may be a useful exercise for the reader to confirm that the following code could be used to implement comparison in a pointer expression if it were not built in.

```
$ fun --m="~&alParPfabbIPWlrBPNQarZPq" --decompile
main = refer conditional(
   field(0,(&,0)),
   conditional(
      field(0,(0,&)),
      conditional(
         recur((&,0),(0,(((&,0),0),(0,(&,0))))),
         recur((&,0),(0,(((0,&),0),(0,(0,&))))),
         constant 0),
      constant 0),
   conditional(field(0,(0,&)),constant 0,constant &))
```

Everything about this example is explained in one previous section or another. Remembering where they are is part of the exercise. Note that the compiler has optimized the code by exploiting the non-strict semantics of the `B` pseudo-pointer to avoid an unnecessary recursive call, thereby allowing the algorithm to terminate as soon as the first discrepancy between the operands is detected.

w – membership This pseudo-pointer tests whether the result returned by its first subexpression is a member of the list or set returned by its second. A true value (`&`) is returned if it is a member, and a false value (`0`) is returned otherwise.

Membership is based on equality as discussed above. The function `~&w` is semantically equivalent to `~&DlrEk` but faster because it is translated to a single virtual machine instruction.

Functional combinators

These two pseudo-pointers correspond to general operations on functions, composition and application.

H – function application The left subexpression is expected to return the function, and the right subexpression is expected to return an argument for the function. The result is obtained by applying the function to the argument. There are no restrictions on types.

This pseudo-pointer is similar to the `R` pseudo-pointer, but more suitable for functions that are not recursively defined and therefore don't need to call themselves. The difference between `H` and `R` is that the latter applies the function to a job containing the function itself along with the argument, whereas `H` applies it just to the argument. Although `H` seems a simpler operation, its virtual machine code is more complicated because it is less frequently used and not directly supported.

O – composition Functional composition is the operation of using the output from one function as the input to another. The composition pseudo-pointer takes two subexpressions representing functions or pointers and feeds the output from the second one into the first one. That is to say, an expression of the form `~&`fg`O` applied to an argument x is equivalent to `~&`f (`~&`g (x)).

The pseudo-pointer for composition rarely needs to be used explicitly because the pointer expression fg`O` is usually equivalent to gf`P`, or just gf where there is no ambiguity. Note that the order is reversed. However, there is one case where they are not equivalent, which is if g is not a pseudo-pointer and not equivalent to an identity pointer such as `~&lrV` or `~&J`. For example, `~&rlXlP` x is not equivalent to `~&l ~&rlX` x and hence not to `~&lrlXO` x

```
$ fun --m="~&rlXlP (('a','b'),('c','d'))" --c
('c','a')
$ fun --m="~&l ~&rlX (('a','b'),('c','d'))" --c
('c','d')
$ fun --m="~&lrlXO (('a','b'),('c','d'))" --c
('c','d')
```

The difference is that `~&rlXlP` refers to the pair of left sides of a reversed pair of pairs, whereas `~&l ~&rlX` refers to the left side of a reversed pair, hence the right side. On the other hand, the equivalence holds in the case of `~&hzXlP`, because `z` is a pseudo-pointer.

```
$ fun --m="~&hzXl <('a','b'),('c','d')>" --c
('a','b')
$ fun --m="~&lhzXO <('a','b'),('c','d')>" --c
('a','b')
$ fun --m="~&l ~&hzX <('a','b'),('c','d')>" --c
('a','b')
```

This function could be expressed simply by `~&h`.

In informal terms, the effect of juxtaposition (or the implicit `P` constructor) where pointers are concerned is to construct the pointer obtained by attaching a copy of the right subexpression to each leaf of the left. Where pseudo-pointers are concerned it is reversed composition. A formal semantics for this operation is best left to compiler developers. A real user of the language is advised to acquire an intuition based on the informal description and to display the decompiled virtual code when in doubt.

To summarize, although this distinction in the meaning of juxtaposition between pointers and pseudo-pointers is usually appropriate in practice, the `O` pseudo-pointer can be used in effect to override it when it isn't, because it represents composition in either case.

2.5 Escapes

There are many more operations that might be worth encoding by pointer expressions than there are letters of the alphabet, even with case sensitivity, and it is useful for compiler developers to have an open ended way of defining more of them. The solution is to express all further pointers and pseudo-pointers by numerical escape codes preceded by the letter `K` in the pointer expression. Because the remaining operations are less frequently required, this format is not too burdensome for normal use.

Recall from Section 2.3.3 that numerical values are also meaningful in pointer expressions as abbreviations for sequences of consecutive `P` constructors. To avoid ambiguity when such a sequence immediately follows an escape code in a pointer, the letter `P` must be used explicitly in such cases. However, a usage such as `K7P2` is acceptable as an abbreviation for `K7PPP`. That is, only the first `P` following the escape code needs to be explicit.

A list of escape codes is shown in Table 2.7. The remainder of this section explains each of them. Because new escape codes are easy for any compiler developer or aspiring compiler developer to add to the language, there is a chance that this list is incomplete for a locally modified version of the compiler. A fully up to date site specific list can be obtained by the command

```
$ fun --help pointers
```

but this output is intended more as a quick reminder than as complete documentation. If undocumented modifications have been made, the likely suspects are resident hackers and gurus. If the output from this command shows that existing operations are missing or numbered differently, then the compiler has been ineptly modified or deliberately forked.

Although these operations are classified by their arity in Table 2.7 and in this section, it is worth pointing out that the arity is more a matter of convention than logical necessity. For example, the transpose operation, `K7`, which reorders the items in a list of lists, is defined as a unary rather than a nullary pseudo-pointer. The subexpression f in a pointer expression of the form f`K7` represents a function with which this operation is composed, as one would expect, but the unary arity means that it is unnecessary and incorrect to write f`K7P` to group them together when used in a larger context, unlike the situation for nullary pointers (cf. Section 2.3.3 and further remarks on page 71). This convention usually saves a keystroke because the transpose is rarely used in isolation, but if it were, then like other unary pseudo-pointers it could be written without a subexpression as `~&K7`, which would be interpreted as `~&iK7`, with the identity deconstructor `i` inferred.

arity	code	meaning
nullary	8	random draw from a list
	22	address enumeration
	27	alternate list items including the head
	28	alternate list items excluding the head
	30	first half of a list
	31	second half of a list
unary	1	all-same predicate
	2	partition by comparison
	6	tree evaluation by `&drPvHo`
	7	transpose
	9	triangle combinator
	11	generalized intersection combinator
	13	generalized difference combinator
	15	distributing bipartition combinator
	17	distributing filter combinator
	20	bipartition combinator
	21	reduction with empty default
	23	address map
	24	partial reification
	33	triangle squared
binary	0	cartesian product
	3	substring predicate
	4	prefix predicate
	5	suffix predicate
	10	generalized intersection by comparison
	12	generalized difference by comparison
	14	distributing bipartition by comparison
	18	subset predicate
	19	proper subset predicate
	25	unzipped partial reification
	26	total reification
	29	merge of lists
	32	map to alternate list items
	34	depth first tree leaf tagging
	35	preorder tree trunk tagging
	36	preorder tree tagging
	37	postorder tree trunk tagging
	38	postorder tree tagging
	39	inorder tree trunk tagging
	40	inorder tree tagging
	41	level order tree leaf tagging
	42	level order tree trunk tagging
	43	level order tree tagging

Table 2.7: pseudo-pointers expressed by escape codes of the form `K`n

2.5.1 Nullary escapes

There is currently two nullary escapes, as explained below.

8 – random list deconstructor

`K8` can be used like a deconstructor to retrieve a randomly chosen item of a list or element of a set. The argument must be non-empty or an exception is raised.

Functional programmers will consider this operation an "impure" feature of the language, because the output is not determined by the input. That is, the result will be different for every run.

```
$ fun --m="~&K8S <'abc','def','ghi'>" --c
'aei'
$ fun --m="~&K8S <'abc','def','ghi'>" --c
'cfh'
```

They will justifiably take issue with the availability of such an operation because it invalidates certain code optimizing transformations. For example, it is not generally valid to factor out two identical programs applying to the same argument if their output is random.

```
$ fun --m="~&K8K8X 'abcdefghijklmnopqrstuvwxyz'" --c
(`r,`f)
$ fun --m="~&K8iiX 'abcdefghijklmnopqrstuvwxyz'" --c
(`q,`q)
```

The first example above performs two random draws from list, but the second performs just one and makes two copies of it.

Despite this issue, the operation is provided in Ursala as one of an assortment of random data generating tactics varying in sophistication. Randomized testing is an indispensable debugging technique, and the code optimization facilities of the compiler are able to recognize randomizing programs and preserve their semantics.

The intent of this operation is that all draws from the list are equally probable. Draws from a uniform distribution are simulated by the virtual machine's implementation of the Mersenne Twister algorithm. For non-specialists, the bottom line is that the quality of randomness is more than adequate for serious simulation work or test data generation, but not for cryptological purposes.

22 – address enumeration

The `K22` pseudo-pointer can be used as a function that takes any list x as an argument and returns a list y of the same length as x, wherein each item is value of the form `(`a`,0)`. The left side a is either `&`, `(`a'`,0)` or `(0,`a'`)`, for an a' of a similar form. Furthermore, each member of y is nested to the same depth, which is the minimum depth required for mutually distinct items of this form, and the items of y are in reverse lexicographic order. Here is an example.

```
$ fun --main="~&K22 'abcdef'" --cast %tL
<
   ((((&,0),0),0),0),
   ((((0,&),0),0),0),
   (((0,(&,0)),0),0),
   (((0,(0,&)),0),0),
   ((0,((&,0),0)),0),
   ((0,((0,&),0)),0)>
```

This function is useful for converting between lists and a-trees, which are a container type explained in Chapter 3. The following example demonstrates this use of it, but should be disregarded on a first reading because it depends on language features documented in subsequent chapters.[1]

```
$ fun --m="^|H(:=^|/~& !,~&)=>0 ~&K22ip 'abcdef'" --c %cN
[
   4:0: `a,
   4:1: `b,
   4:2: `c,
   4:3: `d,
   4:4: `e,
   4:5: `f]
```

27 – alternate list items including the head

The `K27` pseudo-pointer extracts alternating items from a list starting with the head. It is equivalent to the pointer expression `aitBPahPfatt2RCaq`.

```
$ fun --m="~&K27 '0123456789'" --c
'02468'
```

28 – alternate list items excluding the head

The `K28` pseudo-pointer extracts alternating items from a list starting with the one after the head.

```
$ fun --m="~&K27 '0123456789'" --c
'13579'
```

30 – first half of a list

The `K30` pseudo-pointer takes the first $\lfloor n/2 \rfloor$ items from a list of length n.

```
$ fun --m="~&K30S <'123456789','abcd'>" --s
1234
ab
```

[1]The `bash` command `set +H` may be needed to get this example to work.

The algorithms implementing this operation and the following one do not rely on any integer of floating point arithmetic.

31 – second half of a list

The `K31` pseudo-pointer takes the final $\lceil n/2 \rceil$ items from a list of length n.

```
$ fun --m="~&K31S <'123456789','abcd'>" --s
56789
cd
```

Note that if a list is of odd length, the latter part obtained by `K31` will be longer than the first part obtained by `K30`. An easy way of taking the latter $\lfloor n/2 \rfloor$ items instead would be to use `xK30x`. Whether the length of a list x is even or odd, the identity `~&K30K31T` $x \equiv x$ holds.

2.5.2 Unary escapes

In this section, the unary escapes shown in Table 2.7 are explained and demonstrated.

1 – all-same predicate

An escape code of `1` takes a subexpression computing any function or deconstructor at all, applies it to each member of an input list or set, and returns a true value (`&`) if and only if the result is identical in all cases. For an empty argument, the result is always true. If the result of the function in the subexpression differs between any two members, a value of `0` is returned.

A simple example shows the use of this pseudo-pointer to check whether every string in a list contains the same characters, disregarding their order or multiplicity, by using the `s` pseudo-pointer introduced on page 65.

```
$ fun --m="~&sK1 <'abc','cbba','cacb'>" --c
&
$ fun --m="~&sK1 <'abc','cbba','cacc'>" --c
0
```

In the latter example, the third string lacks the letter `b`, and therefore differs from the others.

2 – partition by comparison

The `K2` pseudo-pointer requires a subexpression representing a function applicable to the items of a list, and specifies a function that partitions an input list into sublists whose members share a common value with respect to the function.

This simple example shows how a list of words can be grouped into sublists by their first letter.

Listing 2.4 This is a job for `~&K6`.

```
#import std
#import nat

#comment -[
toy example of a self-describing algebraic expression represented by a
tree of type %sfOZXT]-

nterm =

('+',sum=>0)^: <
   ('*',product=>1)^: <('3',3!)^: <>,('4',4!)^: <>>,
   ('-',difference+~&hthPX)^: <('9',9!)^: <>,('2',2!)^: <>>>
```

```
$ fun --m="~&hK2x <'ax','ay','bz','cu','cv'>" --c
<<'ax','ay'>,<'bz'>,<'cu','cv'>>
```

If the order of the lists in the result is of no concern, the `x` (reversal) operation at the end of `~&hK2x` can be omitted to save time. In this example, it enforces the condition that the lists in the result are ordered by the first occurrence of any of their members in the input. This ordering would maintain the correct representation if the input were a set and the output were a set of sets.

The function represented by the subexpression may be applied multiple times to the same item of the input list in the course of this operation. If the computation of the function is very time consuming and result is not too large, it may be more efficient to compute and store the result in advance for each item, and remove it afterwards. Although the compiler does not automatically perform this optimization, it can be obtained similarly to the example shown below.

```
$ fun --m="~&hiXSlK2rSSx <'ax','ay','bz','cu','cv'>" --c
<<'ax','ay'>,<'bz'>,<'cu','cv'>>
```

The function (in this case only `h`) has its result paired with the each input item by `hiXS`, and the partitioning is performed with respect to the left side of each pair (which consequently stores the function result) by `lK8`. Then the right side of each item of each item of the result (containing the original input data) is extracted by `rSS`.

6 – tree evaluation

A convenient method for representing algebraic expressions over any semantic domain is to use a tree of pairs in which the left side of each pair contains a symbolic name for an operator in the algebra and the right side is its semantic function. The semantic function takes the list of values of the subtrees to the value of the whole tree. This representation is convenient because it allows expressions of arbitrary types to be evaluated by a simple, polymorphic tree traversal algorithm, and also allows the trees to be manipulated easily. It has applications not just for compilers but any kind of symbolic computation.

The value in terms of the embedded semantics for an algebraic expression using this self-describing representation could be obtained by `~&drPvHo`, but is achieved more concisely by `~&iK6` or just `~&K6`. The symbolic names are ignored by this function, but are probably needed for whatever other reason these data structures are being used.

A simple example is shown in Listing 2.4, although it depends on some language features not previously introduced. It is compiled by the command

```
$ fun kdemo.fun --binary
fun: writing `nterm'
```

and the results can be inspected as shown.

```
$ fun nterm --m=nterm --c %sfOXT
('+',188%fOi&)^: <
   ^: (
      ('*',243%fOi&),
      <('3',6%fOi&)^: <>,('4',6%fOi&)^: <>>),
   ^: (
      ('-',515%fOi&),
      <('9',8%fOi&)^: <>,('2',5%fOi&)^: <>>)>
```

This data structure represents the expression $(3 \times 4) + (9 - 2)$ over natural numbers, and can be evaluated as follows.

```
$ fun nterm --m="~&K6 nterm" --c %n
19
```

The expressions in the right sides of the tree nodes in Listing 2.4 are functions operating on lists of natural numbers or constant functions returning natural numbers, and the corresponding expressions in the output above are the same functions displayed in "opaque" format, which shows only their size in quits.[2]

7 – transpose

The `K7` pseudo-pointer takes a subexpression representing a function returning a list of lists and constructs the composition of that function with the transpose operation. The transpose operation takes an input list of lists to an output list of lists whose rows are the columns of the input. For example,

```
$ fun --m="~&iK7 <'abcd','efgh','ijkl','mnop'>" --c
<'aeim','bfjn','cgko','dhlp'>
```

- All lists in the input are required to have the same number of items, or else an exception is raised.
- This operation is useful in numerical applications for transposing a matrix.
- This is a fast operation due to direct support by the virtual machine.

[2] quaternary digits, each equal in information content to two bits

9 – triangle combinator

Escape number 9 is the triangle combinator, which takes a function as a subexpression and operates on a list by iterating the function n times on the n-th item of the list, starting with zero. This small example shows the triangle combinator used on a function that repeats the first and last characters in a string.

```
$ fun --m="~&hizNCTCK9 <'(a)','(b)','(c)','(d)'>" --c
<'(a)','((b))','(((c)))','((((d))))'>
```

11 – generalized intersection combinator

A pointer expression of the form f`K11` represents generalized intersection with respect to the predicate f. Ordinarily the intersection between a pair of lists or sets is the set of members of the left that are equal to some member of the right. The generalization is to allow other predicates than equality.

The subexpression to `K11` is a pseudo-pointer computing a relational predicate. The result is a function that takes a pair of sets or lists, and returns the maximal subset of the left one in which every member is related to at least one member of the right one by the predicate.

Generalized intersection is not necessarily commutative because the predicate needn't be commutative. It doesn't even require both lists to be of the same type. By convention, the result that is returned will always be a subset or a sublist of the left operand.

This example shows generalized intersection by the membership predicate with the `w` pseudo-pointer.

```
$ fun --m="~&wK11 ('abcde',<'cz','xd','ye','wf','ug'>)" --c
'cde'
```

The effect is to return only those letters in the string `'abcde'` that are members of some string in the other operand.

13 – generalized difference combinator

The generalized difference pseudo-pointer, `K13`, is analogous to generalized intersection, above, in that it subtracts the contents of one list from another based on relations other than equality.

The subexpression to `K13` is a pseudo-pointer computing a relational predicate. The result is a function that takes a pair of sets or lists, The function returns a subset of the left one with every member deleted that is related to at least one member of the right one by the predicate, and the rest retained.

A similar example is relevant to generalized difference, where the relational operator is `w` for membership.

```
$ fun --m="~&wK13 ('abcde',<'cz','xd','ye','wf','ug'>)" --c
'ab'
```

The letters `c, `d, and `e, have been deleted because they are members of the strings 'cz', 'xd', and 'ye', respectively.

15 – distributing bipartition combinator

Escape number 15 is used for partitioning a list or set into two subsets according to some data-dependent criterion.

- The subexpression of the pseudo-pointer represents a function computing a binary relational predicate. Call it p.
- The result is a function taking a pair as an argument, whose left side is a possible left operand to p, and whose right side is a list of right operands. Denote the argument by $(x, \langle y_0 \dots y_n \rangle)$.
- The computation proceeds by forming the list of pairs of the left side with each member of the right side, $\langle (x, y_0) \dots (x, y_n) \rangle$.
- The relational predicate p is applied to each pair (x, y_k).
- Separate lists are made of the pairs (x, y_i) for which $p(x, y_i)$ is true and the pairs (x, y_j) for which $p(x, y_j)$ is false.
- The result is a pair of lists $(\langle y_i \dots \rangle, \langle y_j \dots \rangle)$, with the list of right sides of the true pairs the left and the false pairs on the right.

An illustrative example may complement this description. In this example, the relational predicate is intersection, expressed by the c pseudo-pointer, and the function bipartitions a list of strings based on whether they have any letters in common with a given string.

```
$ fun --m="~&cK15 ('abc',<'ox','be','ny','at'>)" --c
(<'be','at'>,<'ox','ny'>)
```

The strings on the left in the result have non-empty intersections with 'abc', making the predicate true, and those on the right have empty intersections.

A more complicated way of solving the same problem without K15 would be by the pointer expression rlrDlrcFrS2XrlrjX. The K15 pseudo-pointer is nevertheless useful because it is shorter and easier to get right on the first try.

17 – distributing filter combinator

This pseudo-pointer behaves identically to the distributing bipartition pseudo-pointer, explained above, except that only the left side of the result is returned (i.e., the list of values satisfying the predicate).

Any pointer expression of the form fK17 is equivalent to fK15lP, but more efficient because the false pairs are not recorded.

The following example illustrates this point.

```
$ fun --m="~&cK17 ('abc',<'ox','be','ny','at'>)" --c
<'be','at'>
```

If only the alternatives are required, they are easily obtained by negating the predicate.

```
$ fun --m="~&cZK17 ('abc',<'ox','be','ny','at'>)" --c
<'ox','ny'>
```

This example uses the pseudo-pointer for negation, explained on page 70.

20 – bipartition combinator

This pseudo-pointer is a simpler variation on the distributing bipartion pseudo-pointer described on page 88. The subexpression f appearing in the context fK20 in a pointer expression can indicate any function computing a unary predicate. The effect is to construct a function taking a list $\langle x_0 \dots x_n \rangle$ and returning a pair of lists $(\langle x_i \dots \rangle, \langle x_j \dots \rangle)$. Each of the x's in the result is drawn from the argument $\langle x_0 \dots x_n \rangle$, but each x_i in the left side satisfies the predicate f, and each x_j in the right side falsifies it. Here is a simple example of the K20 pseudo-pointer being used to bipartition a list of natural numbers according to oddness.

```
$ fun --main="~&hK20 <1,2,3,4,5>" --cast %nLW
(<1,3,5>,<2,4>)
```

This same effect could be achieved by the filtering pseudo-pointer F explained on page 69 and the negation pseudo-pointer Z explained on page 70.

```
$ fun --m="~&hFhZFX <1,2,3,4,5>" --c %nLW
(<1,3,5>,<2,4>)
```

Although semantically equivalent, the latter form is less efficient because it requires two passes through the list and evaluates the predicate twice for each item. It also contains two copies of the code for the same predicate.

21 – reduction with empty default

This pseudo-pointer is useful for mapping a binary operation over a list. The list is partitioned into pairs of consecutive items, the operation is applied to each pair, and a list is made of the results. This procedure is repeated until the list is reduced to a single item, and that item is returned as the result. If the list is initally empty, then an empty value is returned. To be precise, a pointer expression of the form ~&uK21 for a binary pointer operator u is equivalent to ~&iatPfaaitBPahthPuPfatt2RCaqPRahPqB, but more efficient.

This example shows how the union pseudo-pointer (page 77) can be used to form the union of a list of sets of natural numbers.

```
$ fun --m="~&UK21 <{1,2},{3,4},{5},{6,3,1}>" --c %nS
{4,2,6,1,5,3}
```

This example shows a way of concatenating a list of strings.

```
$ fun --m="~&TK21 <'foo','bar','baz'>" --c %s
'foobarbaz'
```

A simpler method of concatenation is by the `~&L` pseudo-pointer (page 65).

23 – address map

The subexpression f in a pointer expression of the form `~&`f`K23` is required to construct a list of (*key*,*value*) pairs wherein each key is an address of the form described in connection with the address enumeration pseudo-pointer on page 82, and further explained in Chapter 3. All keys must be the same size. The result is a very fast function mapping keys to values. Here is an example using the concrete syntax for address type constants.

```
$ fun --m="~&pK23(<5:0,5:1,5:2,5:3,5:4>,'abcde') 5:1" --c
`b
```

24 – partial reification

This pseudo-pointer is similar to the address map pseudo-pointer explained above but doesn't require the keys to be addresses. Here is an example.

```
$ fun --m="(map ~&pK24('abcde','vwxyz')) 'bad'" --c
'wvy'
```

33 – triangle squared

The `K33` pseudo-pointer operates on a list of length n by first making a list of n copies of it, and then applying its operand i times to the i item, numbering from zero. An expression f`K33` is equivalent to `iiDlS`f`K9`, but is implemented using only linearly many applications of the operand f.

```
$ fun --m="~&K33 '0123456789'" --s
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
0123456789
```

Using `K33` with an explicit or implied identity function is equivalent to using `iiDlS`. Using it with the `y` pseudo-pointer (lead of a list) has this effect.

```
$ fun --m="~&yK33 '0123456789'" --s
0123456789
012345678
01234567
0123456
012345
01234
0123
012
01
0
```

2.5.3 Binary escapes

This section explains and demonstrates the binary escape codes listed in Table 2.7. Each of these requires two subexpressions to precede it in the pointer expression where it is used, unless it is at the beginning of the expression, in which case the deconstructors `lr` can be inferred.

0 – cartesian product

For the `K0` pseudo-pointer, both subexpressions are expected to represent functions returning lists or sets, and the result returned by the whole expression is the list of all pairs obtained by taking the left side from the left set and the right side from the right set. Repetitions in the input may cause repetitions in the output.

The following is an example of the cartesian product pseudo-pointer.

```
$ fun --m="~&lyPrtPK0 ('abc',<0,1,2,3>)" --c %cnXL
<(`a,1),(`a,2),(`a,3),(`b,1),(`b,2),(`b,3)>
```

The left subexpression `lyP` by itself would return `'ab'` from this argument, and the right subexpression `rt` would return `<1,2,3>`. The result is therefore the list of pairs whose left side is one of `` `a `` or `` `b ``, and whose right side is one of `1`, `2`, or `3`.

3 – substring predicate

This pseudo-pointer detects whether the result returned by the first subexpression is a substring of the result returned by the second, and returns a true value (`&`) if it is. The operation is polymorphic, so the subexpressions may return either character strings, or lists of any other type.

For a string to be a substring of some other string, it is necessary for the latter to contain all of the characters of the former consecutively and in the same order somewhere within it. Hence, `'cd'` is a substring of `'bcde'`, but not of `'c d'`, `'dc'` or `'c'`. The empty string is a substring of anything.

The following example illustrates this operation with the help of the distributing filter pseudo-pointer explained in the previous section.

```
$ fun --m="~&K3K17 ('cd',<'c d','dc','bcd','cde'>)" --c
<'bcd','cde'>
```

4 – prefix predicate

The prefix pseudo-pointer, `K4`, is a special case of the substring pseudo-pointer explained above, which requires not only the result returned by the first subexpression to be a substring of the result returned by the second, but that it should appear at the beginning, as illustrated by these examples.

```
$ fun --m="~&K4 ('abc','abcd')" --c %b
true
$ fun --m="~&K4 ('abc','ab')" --c %b
false
$ fun --m="~&K4 ('abc','xabc')" --c %b
false
```

5 – suffix predicate

The `K5` pseudo-pointer is a further variation on the substring pseudo-pointer comparable to the prefix, above, except that the substring must appear at the end.

```
$ fun --m="~&K5 ('abc','abcd')" --c %b
false
$ fun --m="~&K5 ('abc','xabc')" --c %b
true
$ fun --m="~&K5 ('abc','ab')" --c %b
false
```

10 – generalized intersection by comparison

The `K10` pseudo-pointer provides an alternative means of specifying generalized intersection to the form discussed on page 87 for the frequently occurring special case of a predicate that compares the results of two separate functions of each side. Any pointer expression of the form `l`f`Pr`g`PEK11` can be expressed alternatively as fg`K10`, thus saving several keystrokes and allowing fewer opportunities for error.

The argument is expected to be a pair of lists. The first subexpression operates on items of the left list, and the second subexpression operates on items of the right list. The result returned by `K10` will be a subset of the left list in which the result of the first subexpression for every member is equal to the result of the second subexpression for some member of the right list.

This simple example shows generalized intersection for the case of a pair of lists of pairs of natural numbers. The criterion is that the left side of a member of the left list has to be equal to the right side of some member of the right list.

```
$ fun --m="~&lrK10 (<(1,2),(3,4)>,<(5,1),(6,7)>)" --c
<(1,2)>
```

That leaves only `(1,2)`, because the left side, `1`, is equal to the right side of `(5,1)`.

12 – generalized difference by comparison

This pseudo-pointer is a binary form of generalized difference, where $f g$`K12` is equivalent to the unary form `l`f`Pr`g`PEK13` discussed on page 87. The predicate compares the results of the two subexpressions f and g applied respectively to the left and the right side of a pair. Because the comparison and relative addressing are implicit, there is no need to write `l`f`Pr`g`PE` when the binary form is used.

A similar example to the above is relevant.

```
$ fun --m="~&lrK12 (<(1,2),(3,4)>,<(5,1),(6,7)>)" --c
<(3,4)>
```

In this example, `l` plays the rôle of f and `r` plays the rôle of g. The pair `(1,2)` is deleted because its left side is the same as the right side of one of the pairs in the other list, namely `(5,1)`.

14 – distributing bipartition by comparison

The binary form of distributing bipartition, expressed by `K14`, performs a similar function to the unary form `K15` explained on page 88. Instead of a single subexpression representing a relational predicate, it requires two subexpressions, each operating on one side of a pair of operands, whose results are compared. Hence, a pointer expression of the form $f g$`K14` is equivalent to `l`f`Pr`g`PEK15`.

An example of this operation is the following, which compares the right side of the left operand to the left side of the each right operand to decide where they belong in the result.

```
$ fun --m="~&rlK14 ((0,1),<(1,2),(3,1),(1,4)>)" --c
(<(1,2),(1,4)>,<(3,1)>)
```

The items in left side of result have `1` on the left, which matches the `1` on the right of `(0,1)`.

16 – distributing filter by comparison

The `K16` pseudo-pointer is similar to `K14`, except that only the list items for which the comparison is true are returned. That is, $f g$`K16` is equivalent to $f g$`K14lP` but more efficient.

```
$ fun --m="~&rlK16 ((0,1),<(1,2),(3,1),(1,4)>)" --c
<(1,2),(1,4)>
```

18 – subset predicate

The `K18` pseudo-pointer computes the subset relation on the results of the two pointers or pseudo-pointers that appear as its subexpressions. The relation holds whenever every member of the left result is a member of the right, regardless of their ordering or multiplicity. If the relation holds, a value of true (`&`) is returned, and otherwise a `0` value is returned. These examples show the simple case of a test for the left side of a pair of sets being a subset of the right.

```
$ fun --main="~&lrK18 ({'b','d'},{'a','b','c','d'})" --c
&
$ fun --main="~&lrK18 ({'b','d'},{'a','b','c'})" --c
0
```

19 – proper subset predicate

The proper subset pseudo-pointer, `K19` tests a similar condition to the subset pseudo-pointer explained above, except that in order for it to hold, it requires in addition that there be at least one member of the right result that is not a member of the left (hence making the left a "proper" subset of the right). These examples demonstrate the distinction.

```
$ fun --main="~&lrK19 ({'b','d'},{'a','b','c','d'})" --c
&
$ fun --main="~&lrK19 ({'b','d'},{'b','d'})" --c
0
$ fun --main="~&lrK18 ({'b','d'},{'b','d'})" --c
&
```

25 – unzipped partial reification

This pseudo-pointer is similar to the partial reification pseudo-pointer explained on page 90, except that each of the subexpressions fg in an expression `~&`fg`K25` is required to construct a list of the same length, with f constructing the list of keys and g constructing the list of values. The result is a fast function mapping keys to values. Here is an example.

```
$ fun --m="(map ~&lrK25('abcde','vwxyz')) 'cede'" --c
'xzyz'
```

26 – total reification

For this pseudo-pointer, the subexpression f in the expression fg`K26` is required to construct a list of (*key*,*value*) pairs, and the subexpression g expresses a function literally. The result is a fast function mapping keys to values, but also able to map any non-key x to `~&`g x. Here is an example in which g is the identiy function.

```
$ fun --m="(map ~&piK26('abcde','vwxyz')) 'bean'" --c
'wzvn'
```

The input `n is not one of the keys `a through `e, so it is mapped to itself in the result. Another choice for g might be N, which would cause any unrecognized input to be taken to an empty result.

29 – merge of lists

The K29 pseudo-pointer takes the lists constructed by each of its two operands and merges them by alternately selecting an item from each. It is not required that the lists have equal length.

```
$ fun --m="~&K29 ('abcde','vwxyz')" --c
'avbwcxdyez'
$ fun --m="~&rlK29 ('abcde','vwxyz')" --c
'vawbxcydze'
```

The expression K27K28K29 is equivalent to the identity function, because the two subexpressions extract alternating items from the argument, which are then merged.

32 – map to alternate list items

A function of the form ~&fgK32 with pointer subexpressions f and g operates on a list by applying ~&f and ~&g alternately to successive items and making a list of the results. That is, a list $\langle x_0, x_1, x_2, x_3 \dots\rangle$ is mapped to $\langle$~&f x_0,~&g x_1,~&f x_2,~&g $x_3 \dots\rangle$. This example shows alternately reversing (x) and taking tails (t) of items in a list of strings.

```
$ fun --m="~&xtK32 <'abc','def','ghi','jkl'>" --s
cba
ef
ihg
kl
```

34 - 43 – tree tagging

The escape codes from 34 through 43 support the simple and often needed operation of uniquely labeling or numbering the nodes in a tree, which crops up occasionally in certain applications and would be otherwise embarrassingly difficult to express in this language.[3]

These pseudo-pointers are meant to appear in a pointer expression such as ~&fgKnn, whose left subexpression f would extract a list from the argument, and whose right subexpression g would extract a tree. The result associated with the combination is a tree having the same shape as the one extracted by g, but with nodes constructed as pairs featuring items from the given list on the left and corresponding nodes from the given tree on the right. In this sense, these operations are similar to that of zipping a pair of lists together to obtain a list of pairs (as described on page 75), with a tree playing the rôle of the right list.

[3]The interested reader is referred to psp.fun in the compiler source distribution for their implementations, or to the output of any command of the form fun --m="~&Knn" --decompile using one of the codes in this range.

Listing 2.5 an m-ary tree of natural numbers in $\langle root \rangle$ ^:<$\langle subtree \rangle \ldots$> format, with 0 for the empty tree

```
#binary+

l = 'abcdefghijklmnopqrstuvw'

t =

204^: <
   242^: <
      134^: <>,
      0,
      184^: <
         289^: <
            753^: <>,
            561^: <>,
            325^: <>,
            852^: <>,
            341^: <>>,
         364^: <>>,
      263^: <>>,
   352^: <
      154^: <
         622^: <
            711^: <>,
            201^: <>,
            153^: <>,
            336^: <>,
            826^: <>>,
         565^: <>>,
      439^: <>,
      304^: <>>>
```

The tree tagging pseudo-pointers operate on trees and lists of any type, but the lexically ordered list of lower case letters and the tree of natural numbers shown in Listing 2.5 are used as a running example. As indicated in previous examples, this notation for trees shows the root on the left of each `^:` operator, and a comma separated list of subtrees enclosed by angle brackets on the right. Leaf nodes have an empty list of subtrees, written <>, and empty subtrees, if any, are represented as null values that can be written as `0`.

By way of motivation, imagine that a graphical depiction of the tree in Listing 2.5 is to be rendered by a tool such as Graphviz,[4] which requires an input specification of a graph consisting of set of vertices and a set of edges. Given a binary file `t` obtained by compiling the code in Listing 2.5, a simple way of extracting the vertices would be like this,

```
$ fun t --m="~&dvLPCo t" --c
<
   204,
   242,
   134,
   184,
   289,
   753,
   561,
   325,
   852,
   341,
   364,
   263,
   352,
   154,
   622,
   711,
   201,
   153,
   336,
   826,
   565,
   439,
   304>
```

and the edges like this.[5]

```
$ fun t --m="~&ddviFlS2DviFrSL3TXor t" --c
<
   (204,242),
   (204,352),
```

[4] `http://www.graphviz.org`
[5] decompilation may be instructive

```
   (242,134),
   (242,184),
   (242,263),
   (184,289),
   (184,364),
   (289,753),
   (289,561),
   (289,325),
   (289,852),
   (289,341),
   (352,154),
   (352,439),
   (352,304),
   (154,622),
   (154,565),
   (622,711),
   (622,201),
   (622,153),
   (622,336),
   (622,826)>
```

However, this approach depends on the assumption of each node in the tree storing a unique value, which might not hold in practice. To address this issue, a unique tag could easily be associated with each node in the list of nodes like this,

```
$ fun t l --m="~&p(l,~&dvLPCo t)" --c
<
   (`a,204),
   (`b,242),
   (`c,134),
   (`d,184),
   (`e,289),
   (`f,753),
   (`g,561),
   (`h,325),
   (`i,852),
   (`j,341),
   (`k,364),
   (`l,263),
   (`m,352),
   (`n,154),
   (`o,622),
   (`p,711),
   (`q,201),
   (`r,153),
```

```
   (`s,336),
   (`t,826),
   (`u,565),
   (`v,439),
   (`w,304)>
```

but doing so brings us no closer to expressing the list of edges unambiguously, which is where tree tagging pseudo-pointers come in. If we try the following,

```
$ fun t l --m="~&K36(l,t)" --c %cnXT
(`a,204)^: <
   (`b,242)^: <
      (`c,134)^: <>,
      ~&V(),
      (`d,184)^: <
         (`e,289)^: <
            (`f,753)^: <>,
            (`g,561)^: <>,
            (`h,325)^: <>,
            (`i,852)^: <>,
            (`j,341)^: <>>,
         (`k,364)^: <>>,
      (`l,263)^: <>>,
   (`m,352)^: <
      (`n,154)^: <
         (`o,622)^: <
            (`p,711)^: <>,
            (`q,201)^: <>,
            (`r,153)^: <>,
            (`s,336)^: <>,
            (`t,826)^: <>>,
         (`u,565)^: <>>,
      (`v,439)^: <>,
      (`w,304)^: <>>>
```

we get tags attached in place on the tree before doing anything else. We could then discard the original node values while preserving the tree structure and guaranteeing uniqueness,

```
$ fun t l --m="~&K36dlPvVo(l,t)" --c %cT
`a^: <
   `b^: <
      `c^: <>,
      ~&V(),
      `d^: <
         ^: (
```

```
            `e,
            <`f^: <>,`g^: <>,`h^: <>,`i^: <>,`j^: <>>),
         `k^: <>>,
      `l^: <>>,
   `m^: <
      `n^: <
         ^: (
            `o,
            <`p^: <>,`q^: <>,`r^: <>,`s^: <>,`t^: <>>),
         `u^: <>>,
      `v^: <>,
      `w^: <>>>
```

and proceed as before to extract the adjacency relation.

```
$ fun t l --m="~&K36dlPvVoddviFlS2DviFrSL3TXor(l,t)" --c
<
   (`a,`b),
   (`a,`m),
   (`b,`c),
   (`b,`d),
   (`b,`l),
   (`d,`e),
   (`d,`k),
   (`e,`f),
   (`e,`g),
   (`e,`h),
   (`e,`i),
   (`e,`j),
   (`m,`n),
   (`m,`v),
   (`m,`w),
   (`n,`o),
   (`n,`u),
   (`o,`p),
   (`o,`q),
   (`o,`r),
   (`o,`s),
   (`o,`t)>
```

The other pseudo-pointer escape codes in the range 34 through 43 differ in the order of traversal or by excluding terminal or non-terminal nodes, as summarized in Table 2.8. The ten alternatives arise as follows.

- A traversal can be either depth first or breadth first.

		depth first		
	breadth first	preorder	postorder	inorder
leaves	`41`	`34`	`34`	`34`
trunks	`42`	`35`	`37`	`39`
both	`43`	`36`	`38`	`40`

Table 2.8: summary of tree tagging pseudo-pointer escape codes

- breadth first traversals tag nodes in level order starting from the root
- depth first traversals apply a contiguous sequence of tags to each subtree

- If it's depth first, it can be either preorder, postorder, or inorder.
  - preorder tags the root first, then the subtrees
  - postorder tags the subtrees first, then the root
  - inorder tags the first subtree first, then the root, and then the remaining subtrees
- Whatever method of traversal is used, it can apply to the whole tree, just the leaves, or just the non-terminal nodes, but depth first traversals applying only to the leaves are independent of the order.

Empty subtrees are almost always ignored, with the one exception being the case of an inorder traversal where the first subtree is empty. Although the empty subtree is not tagged, its presence will cause the root to be tagged ahead of the remaining subtrees, as these examples show.

```
$ fun --m="~&K40('xy','a'^:<'b'^:<>>)" --c %csXT
(`y,'a')^: <(`x,'b')^: <>>
$ fun --m="~&K40('xy','a'^:<0,'b'^:<>>)" --c %csXT
(`x,'a')^: <~&V(),(`y,'b')^: <>>
```

An example of each of each case from Table 2.8 is shown in Tables 2.9 through 2.11. In cases where the number of relevant nodes in `t` is less than the length of the list `l`, the list has been truncated. Truncation is not automatic, and must be done explicitly before the tagging operation is attempted, or a diagnostic message of "`bad tag`" will be reported. However, it is a simple matter to make a list of the leaves or the non-terminal nodes in a tree using the expressions `~&vLPiYo` and `~&vdvLPCBo`, respectively, which can be used to truncate the list of tags by something like this

```
~&llSPrK34(zipt(l,~&vLPiYo t),t)
```

where `zipt` is the standard library function for truncating zip.

whole tree (K36)	just leaves (K34)	just trunks (K35)
(`a,204)^: <	204^: <	(`a,204)^: <
(`b,242)^: <	242^: <	(`b,242)^: <
(`c,134)^: <>,	(`a,134)^: <>,	134^: <>,
0,	0,	0,
(`d,184)^: <	184^: <	(`c,184)^: <
(`e,289)^: <	289^: <	(`d,289)^: <
(`f,753)^: <>,	(`b,753)^: <>,	753^: <>,
(`g,561)^: <>,	(`c,561)^: <>,	561^: <>,
(`h,325)^: <>,	(`d,325)^: <>,	325^: <>,
(`i,852)^: <>,	(`e,852)^: <>,	852^: <>,
(`j,341)^: <>>,	(`f,341)^: <>>,	341^: <>>,
(`k,364)^: <>>,	(`g,364)^: <>>,	364^: <>>,
(`l,263)^: <>>,	(`h,263)^: <>>,	263^: <>>,
(`m,352)^: <	352^: <	(`e,352)^: <
(`n,154)^: <	154^: <	(`f,154)^: <
(`o,622)^: <	622^: <	(`g,622)^: <
(`p,711)^: <>,	(`i,711)^: <>,	711^: <>,
(`q,201)^: <>,	(`j,201)^: <>,	201^: <>,
(`r,153)^: <>,	(`k,153)^: <>,	153^: <>,
(`s,336)^: <>,	(`l,336)^: <>,	336^: <>,
(`t,826)^: <>>,	(`m,826)^: <>>,	826^: <>>,
(`u,565)^: <>>,	(`n,565)^: <>>,	565^: <>>,
(`v,439)^: <>,	(`o,439)^: <>,	439^: <>,
(`w,304)^: <>>>	(`p,304)^: <>>>	304^: <>>>

Table 2.9: three ways of pre-order tagging the tree in Listing 2.5 with letters of the alphabet

whole tree (K43)	just leaves (K41)	just trunks (K42)
(`a,204)^: <	204^: <	(`a,204)^: <
(`b,242)^: <	242^: <	(`b,242)^: <
(`d,134)^: <>,	(`a,134)^: <>,	134^: <>,
0,	0,	0,
(`e,184)^: <	184^: <	(`d,184)^: <
(`j,289)^: <	289^: <	(`f,289)^: <
(`n,753)^: <>,	(`g,753)^: <>,	753^: <>,
(`o,561)^: <>,	(`h,561)^: <>,	561^: <>,
(`p,325)^: <>,	(`i,325)^: <>,	325^: <>,
(`q,852)^: <>,	(`j,852)^: <>,	852^: <>,
(`r,341)^: <>>,	(`k,341)^: <>>,	341^: <>>,
(`k,364)^: <>>,	(`e,364)^: <>>,	364^: <>>,
(`f,263)^: <>>,	(`b,263)^: <>>,	263^: <>>,
(`c,352)^: <	352^: <	(`c,352)^: <
(`g,154)^: <	154^: <	(`e,154)^: <
(`l,622)^: <	622^: <	(`g,622)^: <
(`s,711)^: <>,	(`l,711)^: <>,	711^: <>,
(`t,201)^: <>,	(`m,201)^: <>,	201^: <>,
(`u,153)^: <>,	(`n,153)^: <>,	153^: <>,
(`v,336)^: <>,	(`o,336)^: <>,	336^: <>,
(`w,826)^: <>>,	(`p,826)^: <>>,	826^: <>>,
(`m,565)^: <>>,	(`f,565)^: <>>,	565^: <>>,
(`h,439)^: <>,	(`c,439)^: <>,	439^: <>,
(`i,304)^: <>>>>	(`d,304)^: <>>>	304^: <>>>

Table 2.10: three ways of level-order tagging the tree in Listing 2.5 with letters of the alphabet

order	coverage	
	whole tree (K38/K40)	just trunks (K37/K39)
postorder	(`w,204)^: < (`k,242)^: < (`a,134)^: <>, 0, (`i,184)^: < (`g,289)^: < (`b,753)^: <>, (`c,561)^: <>, (`d,325)^: <>, (`e,852)^: <>, (`f,341)^: <>>, (`h,364)^: <>>, (`j,263)^: <>>, (`v,352)^: < (`s,154)^: < (`q,622)^: < (`l,711)^: <>, (`m,201)^: <>, (`n,153)^: <>, (`o,336)^: <>, (`p,826)^: <>>, (`r,565)^: <>>, (`t,439)^: <>, (`u,304)^: <>>>	(`g,204)^: < (`c,242)^: < 134^: <>, 0, (`b,184)^: < (`a,289)^: < 753^: <>, 561^: <>, 325^: <>, 852^: <>, 341^: <>>, 364^: <>>, 263^: <>>, (`f,352)^: < (`e,154)^: < (`d,622)^: < 711^: <>, 201^: <>, 153^: <>, 336^: <>, 826^: <>>, 565^: <>>, 439^: <>, 304^: <>>>
inorder	(`l,204)^: < (`b,242)^: < (`a,134)^: <>, 0, (`i,184)^: < (`d,289)^: < (`c,753)^: <>, (`e,561)^: <>, (`f,325)^: <>, (`g,852)^: <>, (`h,341)^: <>>, (`j,364)^: <>>, (`k,263)^: <>>, (`u,352)^: < (`s,154)^: < (`n,622)^: < (`m,711)^: <>, (`o,201)^: <>, (`p,153)^: <>, (`q,336)^: <>, (`r,826)^: <>>, (`t,565)^: <>>, (`v,439)^: <>, (`w,304)^: <>>>	(`d,204)^: < (`a,242)^: < 134^: <>, 0, (`c,184)^: < (`b,289)^: < 753^: <>, 561^: <>, 325^: <>, 852^: <>, 341^: <>>, 364^: <>>, 263^: <>>, (`g,352)^: < (`f,154)^: < (`e,622)^: < 711^: <>, 201^: <>, 153^: <>, 336^: <>, 826^: <>>, 565^: <>>, 439^: <>, 304^: <>>>

Table 2.11: four other ways of depth first tagging the tree in Listing 2.5 with letters of the alphabet

2.6 Remarks

Having read this chapter, some readers may be reconsidering their decision to learn the language, perhaps even suspecting it of being an elaborate practical joke in the same vein as `brainf***` or other esoteric languages. However, nothing could be further from the truth, and there is good reason to persevere.

If the material in this chapter seems too difficult to remember, a ready reminder is always available by the command

```
$ fun --help pointers
```

If you have more serious reservations, your documentation engineer can only recommend imagining the view from the top of the learning curve, where you are lord or lady of all you survey. The relentless toil over glue code for every minor text or data transformation is a fading memory. The idea of poring over a thick manual of API specifications full of functions with names like `getNextListElement` and half a dozen parameters seems ludicrous to you. No longer subject to such distractions, your decrees issue effortlessly from your fingers as pseudo-pointer expressions at the speed of thought. They either work on the first try or are easily corrected by a quick inspection of the decompiled code. In view of what you're able to accomplish, it is as if decades of leisure time have been added to your lifespan.

Cool down, big guy. I already told you, you're not my type.
Curdy's last line in *Streets of Fire*

3
Type specifications

The emphasis on type expressions to the tune of a whole chapter may be surprising for an untyped language. In fact, they are no less important than in a strongly typed language, but they are used differently.

- One use already seen in many previous examples is to cast binary data to an appropriate printing format.
- Another important use is for debugging. The nearest possible equivalent to setting a breakpoint and examining the program state is accomplished by a strategically positioned type expression.
- Another use is for random test data generation during development, whereby valid instances of arbitrarily complex data structures can be created to exercise the code and detect errors.
- At the developer's option, type expressions can even specify run-time validation of assertions in production code.
- Type expressions in record declarations can be used to imply default values or initialization functions for the fields without explicitly coding them.
- Certain pattern matching or classification predicates are elegantly expressed in terms of type expressions using tagged unions.
- Type expressions are first class objects that can be stored or manipulated like other data, thereby affording the means for self-describing data structures.

Type expressions also serve the traditional purpose of a formal source level documentation that does not contribute directly to code generation. By being especially concise in this language, they are superbly effective in this capacity because they can be sprinkled liberally and unobtrusively through the code. This benefit often comes freely as a byproduct of their other uses, when they are rephrased as comments after the initial development phase.

The things they don't do are legislation and policy making. Users are very welcome to write badly typed code if they so desire, or to ignore the type system completely. Why does the compiler let them? Aside from the obvious answer that it isn't their nanny, the alternative is to restrict the language to trivial applications with decidable type checking problems, which would drastically curtail its utility. [1]

3.1 Primitive types

Although they are not computationally universal, type expressions are a language in themselves. They have a simple grammar involving nullary, unary, and binary operators using a postfix notation, similarly to pointer expressions described in the previous chapter. Type expressions also provide mechanisms for self-referential structures and for combining literal and symbolic names, all of which require explanation. It is therefore best to postpone the more challenging concepts while dispensing with the easy ones.

Primitive types are the nullary operators in the language of type expressions, and they are the subject of this section. They can be understood independently of the rest of the chapter. As in other languages, primitive types are the basic building blocks of other data structures, and have well defined concrete representations and syntactic conventions. Unlike some other languages, this one includes primitive types whose representations are not necessarily fixed sizes, such as arbitrary precision numbers. Functions are also a primitive type, and are not distinguished by the types of their input or output.

The type expression for a primitive type is of the form `%`t, where t is a single letter, usually lower case. A list of primitive types is shown in Table 3.1. The table also indicates that for some primitive types, a parsing function can be automatically generated, and shows an example instance of the type in the concrete syntax recognized by the compiler and by the parsing function, if any.

3.1.1 Parsing functions

Before moving on to the discussion of specific primitive types, we can take note of the usage of parsing functions. For any of the primitive type expressions `%a`, `%c`, `%e`, `%E`, `%n`, `%q`, `%s`, `%x`, `%v`, or `%z`, there is a corresponding parsing function that can be expressed as `%ap`, `%cp`, *etcetera*, by appending a lower case `p` to the expression. The parsing function takes a list of character strings to an instance of the type.

An example of a parsing function is the following, which transforms a list of character strings containing a decimal number to the standard IEEE floating point representation.

```
$ fun --main="%ep <'123.456'>" --cast %e
1.234560e+02
```

[1] Don't take my word for it. Read the opening soliloquy in any textbook on programming languages and weep.

	type	parser	example
a	address	yes	`15:4924`
b	boolean		`true`
c	character	yes	`` `c ``
e	standard floating point	yes	`4.257736e+00`
E	`mpfr` floating point	yes	`-2.625948E+00`
f	function		`compose(reverse,transpose)`
g	general data		`(5,<'N'>)`
j	complex floating point		`5.089e-01+9.522e+00j`
n	natural number	yes	`21091921548812`
o	opaque		`140%oi&`
q	rational	yes	`-1488159707841741/21667`
s	character string	yes	`'2.I$yTgKs4sqC'`
t	transparent		`(((0,(((&,0),0),(&,&))),0),0)`
v	binary converted decimal	yes	`-21091921548812_`
x	raw data	yes	`-{zxyr{tYGG\sFx<<W{DQVD=B<}-`
y	self-describing		`(-{iUn<}-,-1530566520784/19)`
z	integer	yes	`-21091921548812`

Table 3.1: primitive types

- Parsing functions are useful for operating on contents of text files and command line parameters.
- They pertain only to this set of primitive types, not to type expressions in general.
- When the `p` is appended to a type expression, it is no longer a type expression, but a function, and can be used in any context where a function is appropriate.

3.1.2 Specifics

The remainder of this section discusses each primitive type from Table 3.1 in greater detail.

`a` – Address

The address type is intended as a systematic notation for deconstructing pointers, as discussed in the previous chapter. Recall that a deconstructor is a function that extracts a particular field from an instance of an aggregate type such as a tuple or a list.

Addresses are denoted by a pair of literal decimal constants separated by a colon, with no intervening white space. For an address of the form $n : m$, the number m may range from zero to $2^n - 1$ inclusive.

The numbering convention used for addresses is best motivated by an illustration. In Figure 3.1, a balanced binary tree has a depth of n and leaves numbered from 0 to $2^n - 1$. A tree of this form would be the most appropriate container for a set of data requiring fast (logarithmic time) non-sequential access.

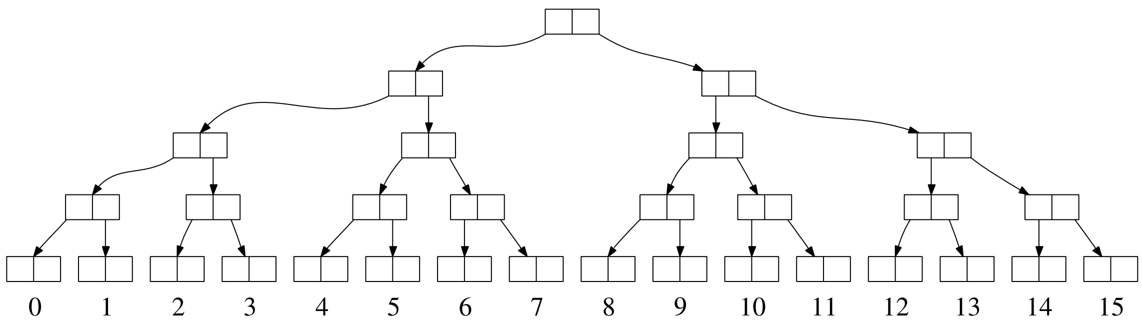


Figure 3.1: a balanced binary tree of depth n with leaves numbered from 0 to $2^n - 1$

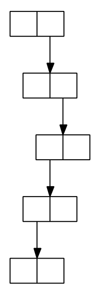


Figure 3.2: descending twice to the right and twice to the left, the address 4:12 points to the twelfth leaf in a tree of depth 4 (cf. Figure 3.1)

The diagram shown in Figure 3.2 depicts the specific address `4:12`. This figure is also a tree, albeit with only one branch descending from each node. There is nevertheless a distinction between whether a branch descends to the left or to the right. The distinction can be seen more clearly by casting the address to a different type.

```
$ fun --main="4:12" --cast %t
(0,(0,((&,0),0)))
```

Here we see a leaf node inside of four nested pairs, located on the right sides of the outer two and the left sides of the inner two.

These observations are true of address type instances in general.

- An address $n : m$ corresponds to a tree with at most one descendent from each node.
- The total number of edges in the tree is n.
- Counting a left branch as 0 and a right branch as 1, the sequence of branches from the root downward expresses m in binary, with the most significant bit first.
- Following the same path from the root of a fully populated balanced binary tree of depth n would lead to the m-th leaf, numbered from 0 at the left.

Note that $n : m$ is metasyntax. In the language n and m must be literal decimal constants.

b – Boolean

The boolean type has two instances, represented as `((),())` and `()` for true and false, respectively. These can also be written as `&` and `0`.

When a value is cast as a boolean type for printing, it will be printed either as `true` or `false`. Strictly speaking these are identifiers rather than literal constants, and will require the standard library `std.avm` or `cor.avm` to be imported in order to be recognized during compilation. However, these libraries are imported automatically by default.

c – Character

The character type has 256 instances represented as arbitrarily chosen nested tuples of `()` on the virtual machine level. The representation is designed to allow lexical comparison of characters by the same algorithm as string comparison, and to ensure that no character representation coincides with that of any numeric type, boolean, or character string.

For printable characters, literal character constants can be expressed by the character preceded by a back quote, as in `` `a``, `` `b`` and `` `c``. For unprintable characters such as controls and tabs, an expression like `~&h skip/9 characters` can be used for the character whose ISO code is 9. The constant `characters` is the list of all 256 characters in lexical order, and is declared in the standard library `std.avm`.

When a value is cast as a character type for printing, the back quote form will be used if the character is printable, but otherwise an expression like `127%cOi&` is generated. The initial decimal number is the ISO code of the character, and the rest of the expression follows the convention used for display of opaque types explained later in this chapter. This latter form can also be used as alternative to the expression involving the `characters` constant described above.

e – Standard floating point

Double precision floating point numbers in the standard IEEE representation are instances of the `e` primitive type.

A full complement of operations on floating point numbers is provided by external libraries optionally linked with the virtual machine, and documented in the `avram` reference manual.

```
$ fun --main="math..sqrt 3." --cast %e
1.732051e+00
```

As noted elsewhere in this manual, the ellipses operator invokes virtual machine library functions by name.

When data are cast to floating point numbers for printing, as above, an exponential notation with seven digits displayed is used by default. Display in user specified formats following C language conventions is also possible through the use of library functions.

```
$ fun --m="math..asprintf('%0.2f',1.23456)" --c
'1.23'
```

When strings are parsed to floating point numbers with the %ep parsing function, it is done by the host machine's C library function strtod, so any C language floating point format is acceptable. However, floating point numbers appearing in program source text must be in decimal, and either a decimal point or an exponent is obligatory to avoid ambiguity with natural numbers. If exponential notation is used, the e must be lower case to distinguish the number from the mpfr type, explained below. There are no implicit conversions between floating point and natural numbers.

Bit level manipulation of floating point numbers is possible for users who are familiar with the IEEE standard, but it is not conveniently supported in the language. A floating point number may be cast losslessly to a list of eight character representations, where each character's ISO code is the corresponding byte in the binary representation.

```
$ fun --m="math..sqrt 3." --c %cL
<
   170%cOi&,
   `L,
   `X,
   232%cOi&,
   `z,
   182%cOi&,
   251%cOi&,
   `?>
```

E – mpfr floating point

On platforms where the virtual machine has been built with support for the mpfr library, a type of arbitrary precision floating point numbers is available in the language, along with an extensive collection of relevant numerical functions, including transcendental functions and fundamental constants. These numbers are not binary compatible with standard floating point numbers, but explicit conversions between them are supported. The mpfr library functions documented in the avram reference manual can be invoked directly using the ellipses operator.

```
$ fun --m="mp..exp 2.3E0" --c %E
9.974182E+00
```

For a number to be specified in this format in a program source text, it should be written in exponential notation with an upper case E to ensure correct disambiguation. That is, 1.0E0 denotes a number in mpfr format, but 1.0e0 and 1.0 denote numbers in standard floating point format. If a number is explicitly parsed by the mpfr parsing function %Ep, then this convention does not apply.

Calculations with numbers in mpfr format do not guarantee exact answers, but in nonpathological cases, the roundoff error can be made arbitrarily small by a suitable choice of precision (up to the available memory on the host). By default, 160 bits of precision are used, which is roughly equivalent to the number of digits shown below.

```
$ fun --m="~&iNC ..mp2str 3.14E0" --s
3.14000000000000000000000000000000000000000000000E+00
```

There are several ways of controlling the precision.

- If a literal `mpfr` constant is expressed in a program source text or in the argument to the `%Ep` parsing function with more than the number of digits corresponding to 160 bit precision, the commensurate precision is inferred.

- Functions returning fundamental constants, such as `mpfr..pi`, or random numbers, such as `mpfr..urandomb`, take a natural number as an argument and return a number with that precision.

- The `mpfr..grow` function takes a pair of operands (x, n) to a copy of x padded with n additional zero bits, for an `mpfr` number x and a natural number n.

- The `mpfr..shrink` function returns a truncated copy.

When the precision of a number is established, all subsequent calculations depending on it will automatically use at least the precision of that number. If two numbers in the same calculation have different precisions, the greater precision is used. Of course, a chain is only as strong as its weakest link, so not all bits in the answer are theoretically justified in such a case.

Low level manipulation of `mpfr` numbers is for hackers only. As a starting point, try casting one to the type `%nbnXXbnXcLXX`.

`f` – Function

Functions are a primitive type in the language, and all functions are the same type. That doesn't mean all functions have the same input and output types, but only that this information is not part of a function's type. This convention allows more flexible use of functions as components of other data structures, such as lists, trees and records, than is possible with more constrained type disciplines. For example, if the language insisted that all functions in a list should have the same input and output types, it would be practically useless for modelling a pipeline or process network as a list of functions.

A value cast to a function type for printing will be expressed in terms of a small set of mnemonics defined in the `cor.fun` library distributed with the compiler (Listing 3.1), whose meanings are documented in the `avram` reference manual. This form very closely follows the underlying virtual machine code representation. Strictly speaking, an understanding of the virtual machine code semantics is not a prerequisite for use of the language. However, it may be helpful for users wishing to verify their understanding of advanced language features by seeing them expressed in terms of more basic ones for small test cases.

The default output format for functions is actually a subset of the language, and in principle could be pasted into a file and compiled, assuming either the `cor` or `std` library is imported. However, functions expressed in this format will be too large and complicated to be of any use as an aid to intuition in non-trivial cases. A useful technique to avoid

Listing 3.1 all programs expressible in the language can be reduced to some combination of these operations

```
#comment -[
This module provides mnemonics for the combinators and built in
functions used by the virtual machine. E.g., compose(f,g) = ((f,g),0)
which the virtual machine interprets as the composition of f and g.

Copyright (C) 2007-2010 Dennis Furey]-

#library+

# constants

false       = 0
true        = &

# first order functions

cat         = (&,&)
weight      = (&,(&,(0,&)))
member      = (&,(&,0))
compare     = &
reverse     = (&,(0,&))
version     = (&,(&,(0,(&,0))))
transpose   = (&,(&,&))
distribute  = ((&,0),0)

# second order functions

fan         = ((((0,&),0),0),(((((&,0),0),(0,&)),0),((0,&),0)))
map         = ((((0,&),0),0),(((((&,0),0),(0,&)),0),(&,0)))
sort        = ((((0,&),0),0),(((((0,&),0),(&,0)),0),((0,&),0)))
race        = (((&,&),((((0,(&,(&,0))),0),0),(0,&))),0)
guard       = (((((&,0),0),(0,(&,0))),0),(0,(0,&)))
recur       = (((((((&,0),0),(0,&)),0),(&,0)),0),(&,0))
field       = (((&,0),0),(0,&))
refer       = (((((((0,&),0),(&,0)),0),(&,0)),0),(&,0))
have        = ((((0,&),0),0),(&,((0,(((&,0),0),(0,&))),&)))
assign      = (((((0,&),0),(&,0)),0),(&,0))
reduce      = ((((0,&),0),0),(((0,&),0),(&,0)))
mapcur      = (((&,&),((((0,(&,(&,0))),0),0),(((0,&),0),(&,0)))),0)
filter      = (((&,&),((((0,(&,&)),0),0),(((0,&),0),(&,0)))),0)
couple      = (((((0,(&,0)),0),(&,0)),0),(0,(0,&)))
compose     = (((0,&),0),(&,0))
iterate     = (((&,&),((((0,(&,&)),0),0),(0,&))),0)
library     = ((((0,&),0),0),(((0,&),0),((0,&),0)))
interact    = ((((0,&),0),0),((((0,(&,0)),0),0),(((((&,0),0),(0,&)),0),(&,0))))
transfer    = (((&,&),((((0,(&,(0,&))),0),0),(0,&))),0)
constant    = (((((&,0),0),(0,&)),0),(&,0))
conditional = (0,(((&,0),(0,(&,0))),(0,(0,&))))
note        = (((&,&),((((0,(&,(&,(0,&)))),0),0),(0,&))),0)
profile     = (((&,&),((((0,(&,(&,&))),0),0),(((0,&),0),(&,0)))),0)
```

being overwhelmed with output when displaying data structures containing functions as components is to use the "opaque" type operator, `O`, explained later in this chapter.

For hackers only: Functions are first class objects in Ursala and can be manipulated meaningfully by anyone taking sufficient interest to learn the virtual machine semantics. A technique that may be helpful in this regard is to transform them to a tree representation of type `%sfOZXT` by way of the disassembly function `%fI`, perform any desired transformations, and then reassemble them by `~&K6` or `~&drPvHo`.

Casual attempts at program transformation are unlikely to improve on the compiler's code optimization facilities, or to add any significant capabilities to the language.[2]

`g` – General data

This type includes everything, but when data are cast to this type for printing, an attempt is made to print them as strings, characters, natural numbers, booleans, or floating point numbers in lists or tuples up to ten levels deep. If this attempt fails, they are printed as raw data, similarly to the `x` type.

- This is the type that is assumed when the `--cast` command line option is used without a parameter.
- If this type is used for a field in a record, it provides a limited form of polymorphism.
- The type inference algorithm used during printing is worst case exponential, and should be used with caution for anything larger than about 500 quits.[3] The worst case arises when the data don't conform to the above mentioned types.

`j` – Complex floating point

Complex numbers are represented in a compatible format with the C language ISO standard and with various libraries, such as `fftw` and `lapack`. That is, they are two contiguously stored IEEE double precision floating point numbers, with the real part first.

When data are cast to complex numbers for printing, the format is always exponential notation with four digits displayed for each of the real part and the imaginary part. However, complex numbers in a program source text may be anything conforming to the syntax ⟨*re*⟩[+|-]⟨*im*⟩[i|j] without embedded spaces. The real and imaginary parts must be C style decimal floating point numbers in fixed or exponential notation, and decimal points are optional. The `i` or `j` must be lower case and must be the last character.

Standard operations on complex numbers are provided by the `complex` library as part of the virtual machine, such as complex division.

```
$ fun --m="c..div(3-4i,1+2j)" --c %j
-1.000e+00-2.000e+00j
```

[2] How's that for throwing down the gauntlet?
[3] quaternary digits; 1 quit = 2 bits

Although there are usually no automatic type conversions in the language, standard floating point numbers are automatically promoted to complex numbers if they are used as an argument to any of the functions in the `complex` library, as this example shows.

```
$ fun --m="c..div(1.,0+1j)" --c %j
0.000e+00-1.000e+00j
```

A complex number can be cast to a list of characters, which will always be of length 16. The first eight characters in the list are the representation of the real part and the second eight are the representation of the imaginary part, as explained in connection with standard floating point types. There should not be any need for low level manipulations of complex numbers under normal circumstances.

```
$ fun --m="2.721-7.489j" --c %cL
<
   248%cOi&,
   `S,
   227%cOi&,
   165%cOi&,
   155%cOi&,
   196%cOi&,
   5%cOi&,
   `@,
   219%cOi&,
   249%cOi&,
   `~,
   `j,
   188%cOi&,
   244%cOi&,
   29%cOi&,
   192%cOi&>
```

`n` – Natural number

Natural numbers are encoded in binary as lists of booleans with the least significant bit first. The representation of the number `0` is the empty list, that of `1` is the list `<&>`, that of two is `<0,&>`, and so on with `<&,&>`, `<0,0,&>`, and `<&,0,&>` *ad infinitum*. The number of bits is limited only by the available memory on the host. There is no provision for a sign bit, because these numbers are strictly non-negative. The most significant bit is always `&`, so the representation of any number is unique. An example of the representation can be seen easily as follows.

```
$ fun --m=1252919 --c %n
1252919
$ fun --m=1252919 --c %tL
<&,&,&,0,&,&,0,0,0,&,&,&,&,0,0,0,&,&,0,0,&>
```

Listing 3.2 hexadecimal printing of naturals by bit twiddling

```
#import std
#import nat

#library+

hex = ||'0'! --(~&y 16); block4; *yx -$digits--'abcdef' pad0 iota16
```

Some applications may take advantage of this representation to perform bit level operations. For example, the function `~&iNiCB` doubles any natural number, the function `~&itB` performs truncating division by two, and the function `~&ihB` tests whether a number is odd. The check for non-emptiness can be omitted to save time if it is known that the number is non-zero.

```
$ fun --m="~&NiC 1252919" --c %tL
<0,&,&,&,0,&,&,0,0,0,&,&,&,&,0,0,0,&,&,0,0,&>
$ fun --m="~&NiC 1252919" --c %n
2505838
```

It is also possible to treat natural numbers as an abstract type by using only the functions defined in the `nat` library to operate on them.

```
$ fun --m="double 1252919" --c %n
2505838
```

Natural numbers expressed in decimal in a source text are converted to this representation by the compiler. Anything cast as a natural number is printed in decimal. However, it is always possible to print them in other ways, such as hexadecimal as shown in Listing 3.2. Some language features used in this listing will require further reading.

o – Opaque

This type includes everything, and is used mainly as the type of an untyped field in a record or other data structure. When a value is displayed as an opaque type, no information about it is revealed except its size measured in quarternary digits (quits).[4]

```
$ fun --m="'allworkandnoplaymakesjackadullboy'" --c %o
320%oi&
```

The number in the prefix of the expression is the size, and the rest of it is the notation used to indicate an opaque type instance.

This notation can also be used in a source text to represent arbitrary random data of the given size, which will be evaluated differently for every compilation.

[4] Due to some overhead inherent in the use of a list representation, a natural number requires one quit for each `0` bit and two quits for each `&` bit.

```
$ fun --m="16%oi&" --c %o
16%oi&
$ fun --m="16%oi&" --c %t
((((&,0),0),(0,((&,0),0))),((0,(0,&)),(&,&)))
$ fun --m="16%oi&" --c %t
(0,(0,(0,(((0,&),(&,&)),(((&,0),0),(0,&))))))
```

This usage is intended mainly for generating test data. Obviously, if data cast as opaque are displayed and copied into a source text to be recompiled, there can be no expectation of recovering the original data unless the size is zero or one.

q – Rational

Exact rational arithmetic involving arbitrary precision rational numbers is possible using the `q` type and associated functions in the `rat` library distributed with the compiler.

Rational numbers are represented as a pairs of integers, with one for the numerator and one for the denominator. Only the numerator may be negative. This example shows a rational number case as a natural (`%q`) type, and as pair of integers (`%zW`).

```
$ fun --main="-1/2" --cast %q
-1/2
$ fun --main="-1/2" --cast %zW
(-1,2)
```

As the above example shows, standard fractional notation is used for both input and output. There may be no embedded spaces, and the numerator and denominator must be literal constants (not symbolic names). The compiler will automatically convert rational numbers to simplest terms to ensure a unique representation.

```
$ fun --m="3/9" --c %q
1/3
```

The algorithm used for simplifying fractions does not employ any sophisticated factorization techniques and will be time consuming for large numbers.

Although rational numbers may be helpful for theoretical work because the results are exact, they are unsuitable for most practical numerical applications because the amount of memory needed to represent a number roughly doubles with each addition or multiplication. The arbitrary precision floating point type (`E`) implemented by the `mpfr` library is a more appropriate choice where high precision is needed.

s – Character string

Used in many previous examples but not formally introduced, the character string type is appropriate for textual data, and is expressed by the text enclosed in single quotes.

Character strings are (almost) semantically equivalent to lists of characters, represented as described in connection with the `c` type.

```
$ fun --m="'abc'" --c %s
'abc'
$ fun --m="'abc'" --c %cL
<`a,`b,`c>
```

The only difference between character strings and lists of characters (aside from cosmetic differences in the printed format) is that strings may contain only printable characters, which are those whose ISO codes range from 32 to 126 inclusive.

Literal quotes The convention for including a literal quote within a string is to use two consecutive quotes.

```
$ fun --m="'I''m a string'" --c
'I''m a string'
```

As shown above, this convention is followed in the output of a quoted string as well, although the extra quote is not really stored in the string. A bit of extra effort shows the raw data.

```
$ fun --main="<'I''m a string'>" --show
I'm a string
```

As one might gather, the `--show` command line option dumps the value of the main expression to standard output, provided that is a list of character strings.

Dash bracket notation On a related note, an easier way of expressing a list of character strings is by the dash bracket notation.

```
$ fun --m="-[I'm a list of strings]-" --show
I'm a list of strings
```

An advantage of this notation is that it allows literal quotes, and in a source text (as opposed to the command line) it may span multiple lines (as shown with `#comment` directives in previous source listings).

A further advantage of the dash bracket notation is that it can be nested in matched pairs like parentheses.

```
$ fun --m="-[I'm -[ <'nested'> ]- in it]-" --show
I'm nested in it
```

Although it's of no benefit in this small example, the advantage of nested dash brackets in general is that the expression inside the inner pair is not required to be a literal constant. It can be any expression that evaluates to a list of character strings. That includes those containing symbolic names, more dash brackets, and arbitrary amounts of white space.

It is also possible to have multiple instances of nested dash brackets inside a single enclosing pair, as shown below.

```
$ fun --m="-[I'm -[<'nested'>]- in-[ <'to'>]- it]-" --s
I'm nested into it
```

Note that the white space inside the second nested pair is not significant.

`t` – Transparent

The transparent type includes everything, and is useful only when the precise virtual machine representation of the data is of interest.

If data are cast to a transparent type for printing, they will be displayed as nested pairs of `0` and `&`. For example, if someone really wanted to know how a character string is represented, the answer could be obtained as shown.

```
$ fun --m="'hal'" --c %t
((&,((0,&),(0,&))),((&,(&,&)),((&,((0,(0,(0,&))),0)),0)))
```

More practical uses are for displaying pointers or virtual machine code when debugging takes a particularly ugly turn. However, this output format quickly grows unmanageable with data of any significant size.

`v` – Binary converted decimal

This type provides an alternative representation for integers as a $(sign, magnitude)$ pair, where the magnitude is a list of natural numbers (type `%n`) each in the range 0 through 9, specifying the decimal digits of the number being represented, with the least significant digit at the head. The sign is a boolean value, equal to `0` for zero and positive numbers and `&` for negatives.

BCD numbers are written with a trailing underscore to distinguish them from naturals (`%n`) and integers (`%z`). For example, these are BCD numbers

```
-28093_ 9289_ -2939_ -46132_ -7691_
```

unlike these, which are integers and naturals.

```
-14313 54188 61862 -196885 84531
```

The type identifier `%v` has no mnemonic significance.

Similarly to the integer and natural types, the size of BCD numbers is limited only by the available host memory. However, for calculations involving numbers in the hundreds of digits or more, there may be a moderate performance advantage in using the BCD representation, especially if the results are to be displayed in decimal. Mathematical operations on numbers are provided by the `bcd` library distributed with the compiler.

`x` – Raw data

This type is similar to the transparent type in that it includes everything, but the display format is meant to be more concise than human readable, by packing three quits into each character.

```
$ fun --m="'dave'" --c %x
-{{cucl<Sb]><}-
```

The format of the text between the leading `-{` and trailing `}-` is the same one used by the virtual machine for binary files, and is documented in the `avram` reference manual. This fact could be exploited to paste the data from a binary file into a source text and compile it.[5]

The use for this type is also in debugging, when the value of some data structure displayed in the course of a run or a crash dump needs to be captured losslessly for further analysis but its exact representation is either unknown or not relevant.

y – Self-describing

An instance of the self-describing type consists of a pair whose left side is a compressed binary representation of a type expression and whose right side is an instance of the type specified by the expression. Data in this format can be cast as `%y` without reference to the base type and displayed correctly, because the necessary information about their type is implicit. The compressed type expression is displayed in raw format along with the data so as to be machine readable.

Self describing types are a more sophisticated alternative to general types `%g`, because they may include records or other complex data structures and be printed accordingly. They are useful for binary files in situations when it might otherwise be difficult to remember the types of their contents. They may also afford a rudimentary form of support for a (not recommended) programming style in which data are type-tagged and functions are predicated on the types of their arguments (an idea dating from the sixties and later revived by the object oriented community). This approach would require the developer to become familiar with the compiler internals.

The right way to construct an instance of a self-describing type is to use a type expression with `Y` appended, for example, `%jY` for a self describing complex number. Semantically, the expression ending in `Y` is a function rather than a type expression. It is meant to be applied to an argument of the base type, (e.g., a complex number) and it will return a copy of the argument with the compressed type expression attached to it. This result thereafter can be treated as a self-describing type instance.

```
$ fun --m="%jY 2-5j" --c %y
(-{iUF<}-,2.000e+00-5.000e+00j)
```

For reasons of efficiency, functions of the form %*t*`Y` perform no check that their arguments are actually a valid instance of the type %*t*, so it is possible to construct a self-describing type instance that doesn't describe itself and will cause an error when it is cast as self describing.[6]

```
$ fun --main="%cY 0" --c %xgX
(-{iU^\}-,0)
$ fun --main="%cY 0" --c %y
fun: invalid text format (code 3)
```

[5] surely a winning strategy for obfuscated code competitions

[6] Don't do this unless you're an academic who's hard pressed for an example to warn people about the dangers of non-type-safe languages.

The above error occurs because `0` is not a valid character instance.

For a correctly constructed self describing type instance, the original data can always be recovered using the ordinary pair deconstructor function, `~&r`.

```
$ fun --m="~&r (-{iUF<}-,2.000e+00-5.000e+00j)" --c %j
2.000e+00-5.000e+00j
```

z – Integer

The integer type (`%z`) pertains to numbers of the form $\cdots -2, -1, 0, 1, 2 \ldots$. For non-negative integers, the representation is the same as that of natural numbers (page 115), namely a list of bits with the least significant bit first, and a non-zero most significant bit. Negative integers are represented as the magnitude in natural form with a zero bit appended. The following examples show a positive and a negative integer cast as integer types (`%z`) and as lists of bits (`%tL`).

```
$ fun --main="13" --cast %z
13
$ fun --main="-13" --cast %z
-13
$ fun --main="13" --cast %tL
<&,0,&,&>
$ fun --main="-13" --cast %tL
<&,0,&,&,0>
```

3.2 Type constructors

As a matter of programming style, most applications can benefit from the use of aggregate types and data structures. The way of building more elaborate types from the primitive types documented in the previous section is by type constructors. Type constructors in this language fall into two groups, which are binary and unary. The binary type constructors are explained first because there are fewer of them and they're easier to understand.

3.2.1 Binary type constructors

One way of using a binary type constructor in a type expression is by writing something of the form %uvT, where u and v are either primitive types or nested type expressions, and T is the binary type constructor. Other alternatives are documented subsequently, but this usage suffices for the present discussion. In this context, u and v are considered the left and right subexpressions, respectively.

The binary type constructors in the language are listed in Table 3.2, and explained below.

		example	
	constructor	expression	instance
`A`	assignment	`%seA`	`'z@Ec+': 2.778150e+00`
`D`	dual type tree	`%qjD`	`-15008/1349^: <6.924+3.646j^: <>>`
`U`	free union	`%EcU`	`` `Y ``
`X`	pair	`%abX`	`(9:275,false)`

Table 3.2: binary type constructors

`A` – Assignment

The assignment type constructor `A` pertains to data that are expressed according to the syntax ⟨*name*⟩`:` ⟨*meaning*⟩ or `~&A(`⟨*name*⟩`,`⟨*meaning*⟩`)` as documented in the previous chapter. The left subexpression u in a type expression of the form `%`uv`A` is the type of the ⟨*name*⟩ field, and the right subexpression v is the type of the ⟨*meaning*⟩ field. Although the pointer constructor `~&A` uses the same letter as the related type constructor, they don't coincide for all other types.

The example in Table 3.2 demonstrates the case of a type expression describing assignments whose name fields are character strings and whose meaning fields are floating point numbers.

`D` – Dual type tree

The `D` type constructor pertains to trees whose non-terminal nodes are a different type from the terminal nodes. In a type expression of the form `%`uv`D`, the type of the non-terminal nodes is u, and the type of the terminal or leaf nodes is v.

The example in Table 3.2 shows a tree using the notation

⟨*root*⟩`^:` <[⟨*subtree*⟩[`,`⟨*subtree*⟩]*]>

where the `^:` operator joins the root to a list of subtrees, each of a similar form, in a comma separated sequence enclosed by angle brackets. For a non-terminal node, the list of subtrees is non-empty, and for a terminal node, it is the empty list, <>.

We therefore have the type expression `%qjD` for trees whose non-terminal nodes are rational numbers, and whose terminal nodes are complex numbers. Accordingly, one instance of this type is a tree whose root node is the rational number `-15008/1349`, and that has one leaf node, which is the complex number `6.924+3.646j`.

`U` – Free union

The free union of two types u and v, given by the expression `%`uv`U`, includes all instances of either type as its instances. When a value is cast as a free union, the appropriate syntax to display it is automatically inferred from its concrete representation.

Free unions therefore work best when the types given by the subexpressions have disjoint sets of instances. In many cases, this condition is easily met. The concrete representations of characters, strings, and rationals are mutually disjoint, and therefore always allow unions between them to be disambiguated correctly. Naturals and booleans are disjoint from characters and rationals. Floating point numbers, complex numbers, and `mpfr` numbers are also mutually disjoint, and disjoint from all of the above except strings. Addresses are disjoint from everything except for the degenerate case `0:0`, which coincides the boolean value of `true`. Tuples, assignments, and records in which the corresponding fields are disjoint are necessarily also disjoint. This fact can be used to effect tagged unions, but a better way is documented subsequently.

If the types in a free union are not mutually disjoint, priority is given to the left subexpression. For example, a free union between naturals and strings will interpret the empty tuple `()` as either the empty string `''` or the number zero depending on which subexpression is first.

```
$ fun --m="()" --c %nsU
0
$ fun --m="()" --c %snU
''
```

`X` – Pair

The `X` type constructor pertains to values expressed by the syntax `(`⟨*left*⟩`,`⟨*right*⟩`)`. The left subexpression u in a type expression of the form `%`uv`X` is the type of the ⟨*left*⟩ field, and the right subexpression v is the type of the ⟨*right*⟩ field.

The example shows the expression `%abX`, representing pairs whose left sides are addresses and whose right sides are booleans. We therefore have `(9:275,false)` as an instance of this type.

Similarly to assignment types, the same letter, `X`, is used for pointer expressions as in `~&lrX`. The meanings are related but in general pointers have a distinct set of mnemonics from type expressions.

3.2.2 Unary type constructors

The remaining type constructors used in the language are unary type constructors, which specify types that are derived from a single subtype. For the examples in this section, type expressions of the form `%`uT suffice, where T is a unary type constructor and u is an arbitrary type expression, whether primitive or based on other constructors.

A list of unary type constructors is shown in Table 3.3. Each of them is explained in greater detail below.

`G` – Grid

The `G` type constructor specifies a type of data structure that can be envisioned as shown in Figure 3.3. The data are stored at the nodes depicted as dots, and a relationship among

		example	
constructor		expression	instance
G	grid	%nG	<[0:0: 134628^: <7:10>],[7:10: 3^: <>]>
J	job	%cJ	~&J/44%fOi& `2
L	list	%bL	<true,false,true>
N	a-tree	%cN	[10:145: `C,10:669: `I,10:905: `A]
O	opaque	%fO	2413%fOi&
Q	compressed	%sQ	%Q('zQPGJ26')
S	set	%sS	{'Pfo','PzHYgmq','We&*'}
T	tree	%eT	3.262893e+00^: <-9.536086e+00^: <>>
W	pair	%EW	(7.290497E+00,-9.885898E+00)
Z	maybe	%qZ	()
m	module	%qm	<'zu': 5/9,'aj': 60/1,'Pj': -1/24>

Table 3.3: unary type constructors

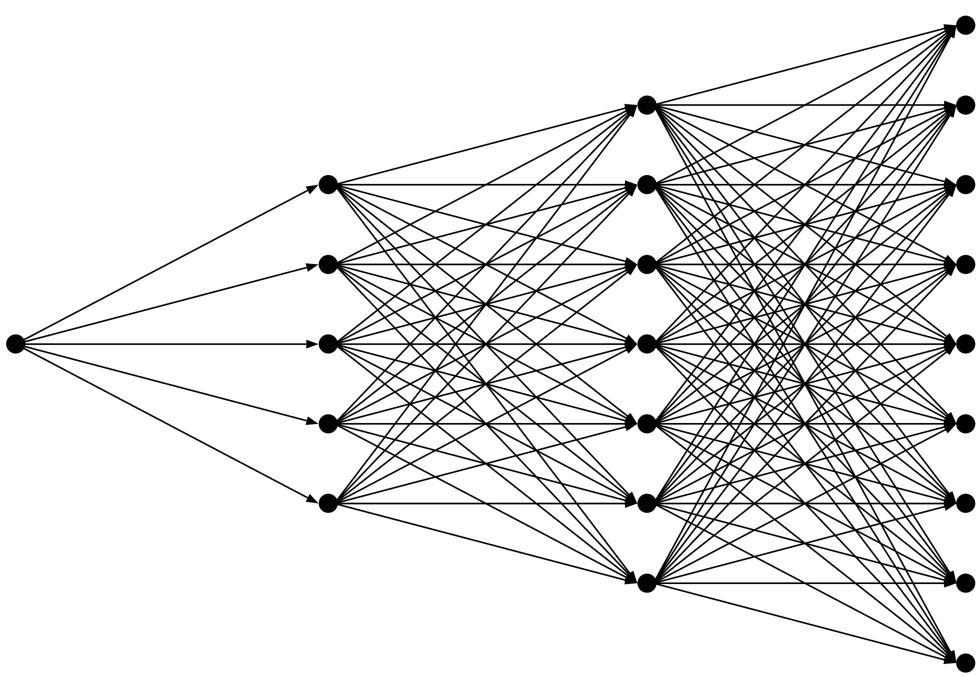


Figure 3.3: an ensemble of trees with subtrees shared among them

them is encoded by the connections of the arrows.

- The number of nodes and the pattern of connections varies from one grid instance to another. Not all possible connections nor any regular pattern is required.
- A common feature of all grids is a partition among the nodes by levels, such that connections exist only between nodes in consecutive levels. The number of levels varies from one grid instance to another.
- Every node in the grid is reachable from a node in the first level, shown at the left, which may contain more than one node.

This structure therefore can be understood as either a restricted form of a rooted directed graph, or as an ensemble of trees with a possibility of vertices shared among them. The purpose of such a representation is to avoid duplication of effort in an algorithm by allowing traversal of a shared subtree to benefit all of its ancestors. In some situations, this optimization makes the difference between tractability and combinatorial explosion. Algorithms exploiting this characteristic of the data structure are facilitated by functional combining forms defined in the `lat` library distributed with the compiler. See Section 1.2.3 for a simple example of a practical application.

One of the few advantages of an imperative programming paradigm is that structures like these have a very natural representation wherein each node stores a list of the memory locations of its descendents. When a shared node is mutably updated, the change is effectively propagated at no cost. A similar effect can be simulated in the virtual machine's computational model as follows.

- An address (of the primitive type `%a`) is arbitrarily assigned to each node.
- Each level of the grid is represented as a separate balanced binary tree (or as balanced as possible) of the form shown in Figure 3.1, with the nodes stored in the leaves. The path from the root to any leaf is encoded by its address, so its address is not explicitly stored.
- Each node contains a list of the addresses (in the above sense) of the nodes it touches in the next level, which belong to a separate address space.
- The following concrete syntax is used to summarize all of this information.

```
<
   [
      ⟨local address⟩: ⟨node⟩^: <⟨descendent's address⟩ ...>,
      ...],
   ⋮
   [
      ⟨local address⟩: ⟨node⟩^: <>,
      ...]>
```

Table 3.3 shows a small example of a grid of strings using this syntax, where there are two levels and only one node in each level. A larger example using a different type (`%sG`) is the following.

```
<
   [0:0: 'egi'^: <8:67,8:144,8:170,8:206>],
   [
      8:206: 'def'^: <10:648,10:757,10:917,10:979>,
      8:170: 'fgh'^: <10:342,10:345,10:757,10:917>,
      8:144: 'acf'^: <10:342,10:757,10:978,10:979>,
      8:67: 'deh'^: <10:345,10:648,10:917,10:978>],
   [
      10:979: 'chj'^: <4:0,4:9,4:10,4:15>,
      10:978: 'cgj'^: <4:3,4:9,4:11,4:15>,
      10:917: 'efi'^: <4:0,4:9,4:11,4:15>,
      10:757: 'adi'^: <4:3,4:9,4:10>,
      10:648: 'abh'^: <4:0,4:10,4:11>,
      10:345: 'cij'^: <4:0,4:3,4:11,4:15>,
      10:342: 'aeg'^: <4:3,4:10,4:11>],
   [
      4:15: 'bdi'^: <>,
      4:11: 'ehi'^: <>,
      4:10: 'acd'^: <>,
      4:9: 'ghj'^: <>,
      4:3: 'abc'^: <>,
      4:0: 'aei'^: <>]>
```

Note that the addresses in the list at the right of each node are relative to the address space of the succeeding level, and that the pattern of connections is irregular.

A few other points about grid types should be noted.

- A type of the form `%`*t*`G` is similar to a type `%`*t*`TNL` using constructors explained later in this section, but not identical because the effect of shared subtrees is not captured by the latter. A type `%`*t*`aLANL` is in some sense "upward compatible" with `%`*t*`G`, but is displayed differently and implies no relationships among the addresses.
- Although grids can have multiple root nodes, the combinators defined in the `lat` library work only for grids with a single root.
- Grids of types that include everything (such as `%g`, `%o`, `%t`, and `%x`) and that also have multiple root nodes might defeat the algorithm used to display them by the `--cast` option, because there is insufficient information to infer the grid topology efficiently from the concrete representation. They can still be used in practice if this information is known and maintained extrinsically (or by inserting a unique root node).
- Badly typed or ambiguous grids that don't cause an exception may be displayed with empty levels. Unreachable nodes are not displayed, but they can be detected as type

errors by debugging methods explained subsequently, or displayed by the upward compatible type cast mentioned above.

- Compared to the grid type constructor, the rest are easy.

`J` – Job

As explained in the previous chapter, the style of anonymous recursion supported by the virtual machine and related pseudo-pointers implies that a function of the form `refer` f applied to an argument x evaluates to f`(~&J(`f`,`x`))`, where the expression `~&J(`f`,`x`)`, called a "job", contains a copy of the recursive function (without the `refer` combinator) along with the original argument, x. Jobs are represented as pairs with the function on the left and the argument on the right, but it is more mnemonic to regard them as a distinct aggregate type with its own constructor and deconstructors, `~&J`, `~&f`, and `~&a`, respectively.

Although a job has two fields, one of them, `~&f`, is always a function, and functions in Ursala are primitive types. The type of a job is therefore determined by the type of the other field, `~&a`. The job type constructor is consequently a unary type constructor, whose base type is that of the argument field.

When a value `~&J(`⟨*function*⟩`,`⟨$argument$⟩`)` is cast as a job type `%`t`J` for printing, the output is of the form

`~&J/`⟨*size*⟩`%fOi&` ⟨*text*⟩

where ⟨*size*⟩ is a decimal number giving the size of the function measured in quits, and ⟨*text*⟩ is the display of the argument cast as the type `%`t. The opaque display format is used for the function field because the explicit form is likely to be too verbose to be helpful.

`L` – List

The list type constructor, `L`, pertains to the simplest and most ubiquitous data structure in functional languages, wherein members are stored to facilitate efficient sequential access. As shown in many previous examples, the concrete syntax for a list in Ursala consists of a comma separated sequence of items enclosed in angle brackets.

<$item_0$`,`$item_1$`,` ... $item_n$>

There is also a concept of an empty list, which is expressed as `<>`. As explained in the previous chapter, lists can be constructed by the `~&C` data constructor, and non-empty lists can be deconstructed by the `~&h` and `~&t` functions.

It is customary for all items of a list to be of the same type. The base type t in a type expression of the form `%`t`L` is the type of the items. A list cast to this type is displayed with the items cast to the type `%`t.

The convention that all items should be the same type, needless to say, is not enforced by the compiler and hence easy to subvert. However, it is just as easy and more rewarding to think in terms of well typed code when a heterogeneous list is needed, by calling it a list of a free unions.

```
$ fun --m="<1,'a',2,3,'b'>" --c %nsUL
<1,'a',2,3,'b'>
```

Free unions are explained in Section 3.2.1.

Because there is no concept of an array in this language, the type `%eL` (lists of floating point numbers) is often used for vectors, and `%eLL` (lists of lists of floating point numbers) for (dense) matrices. The virtual machine interface to external numerical libraries involving vectors and matrices, such as `fftw` and `lapack`, converts transparently between lists and the native array representation. The `avram` reference manual also documents representations for sparse and symmetric matrices as lists, along with all calling conventions for the external library functions.

N – A-tree

Although there are no arrays in Ursala, there is a container that is more suitable for non-sequential access than lists, namely the a-tree, mnemonic for addressable tree.

The concrete syntax for an a-tree is a comma separated sequence of assignments of addresses to data values, enclosed in square brackets, as shown below.

$$
\begin{array}{l}
[\\
\quad a_0\texttt{:} \quad x_0\texttt{,} \\
\quad a_1\texttt{:} \quad x_1\texttt{,} \\
\quad \ldots \\
\quad a_n\texttt{:} \quad x_n\texttt{]}
\end{array}
$$

The addresses a_i follow the same syntax as the primitive address type, `%a`, namely a colon separated pair of literal decimal constants, $n\!:\!m$, with m in the range 0 through $2^n - 1$. For a valid a-tree, all addresses must have the same n value. The data x_i can be of any type.

A type expression of the form `%`t`N` describes the type of a-trees whose data values are of the type `%`t. An example of an a-tree of type `%qN`, containing rational numbers, expressed in the above syntax, would be the following.

```
[
   8:1: 0/1,
   8:22: 1569077783/212,
   8:24: 2060/1,
   8:76: -21/1,
   8:140: 9/3021947915,
   8:187: -198733/2,
   8:234: 10/939335417423]
```

The crucial advantage of an a-tree is that all fields are readily accessible in logarithmic time by way of a single deconstruction operation.

```
$ fun --m="~2:0 [2:0: 'foo',2:1: 'bar',2:2: 'baz']" --c
```

```
'foo'
$ fun --m="~2:1 [2:0: 'foo',2:1: 'bar',2:2: 'baz']" --c
'bar'
$ fun --m="~2:2 [2:0: 'foo',2:1: 'bar',2:2: 'baz']" --c
'baz'
```

As shown above, the deconstructor function is given simply by the address of the field as it is displayed in the default syntax.

This efficiency is made possible by the representation of a-trees as nested pairs.

```
$ fun --m="[2:0: 'foo',2:1: 'bar',2:2: 'baz']" --c %sWW
(('foo','bar'),'baz','')
```

This output is actually a sugared form of `(('foo','bar'),('baz',''))`, which shows more clearly that all data values are nested at the same depth, making them all equally accessible.

```
$ fun --m="(('foo','bar'),('baz',''))" --c %sN
[2:0: 'foo',2:1: 'bar',2:2: 'baz']
```

Moreover, the addresses aren't explicitly stored at all, but are an epiphenomenon of the position of the corresponding data within the structure. The deconstruction operation by the address works because of the representation of address types as shown in Figure 3.2, and the semantics of deconstruction operator, `~`.

The formatting algorithm for a-trees will infer the minimum depth consistent with valid instances of the base type. If the base type is a free union, there is a possibility of ambiguity. For example, if the data can be either strings or pairs of strings, the expression above is displayed differently.

```
$ fun --m="[2:0: 'foo',2:1: 'bar',2:2: 'baz']" --c %ssWUN
[1:0: ('foo','bar'),1:1: ('baz','')]
```

A few further remarks about a-trees:

- Other language features such as the assignment operator, `:=`, are useful for manipulating a-trees, and will require further reading. This is a pure functional combinator despite its connotations.
- There is no reliable way to distinguish between unoccupied locations in an a-tree and locations occupied by empty values. Neither is displayed. Attempts to extract the former will sometimes but not always cause an invalid deconstruction exception. A-trees are best for base types that don't have an empty instance, such as tuples and records.
- Experience is the best guide for knowing when a-trees are worth the trouble. Large state machine simulation problems or graph searching algorithms are obvious candidates. An a-tree of states or graph nodes each containing an adjacency list storing the addresses of its successors might allow fast enough traversal to compensate for the time needed to build the structure.

O – Opaque

The opaque type constructor can be appended to any type `%`t to form the opaque type `%`t`O`. These two types are semantically equivalent but displayed differently when printed as a result of the `--cast` command line option.

Opaque syntax When a value is cast as type `%`t`O`, for any type expression t (other than `c`), it is displayed in the form ⟨*size*⟩`%`t`Oi&` where ⟨*size*⟩ is a decimal number giving the size of the data measured in quits, and t is the same type expression appearing in the cast `%`t`O`. For example,

```
$ fun --m="<1,2,3,4>" --c %nLO
17%nLOi&
$ fun --m="2.9E0" --c %EO
186%EOi&
$ fun --m=successor --c %fO
40%fOi&
```

Opaque semantics The reason for the unusual form of these expressions is that it has an appropriate meaning implied by the semantics of the operators appearing in them (which are explained further in connection with type operators). The expressions could be compiled and their value would be consistent with the type and size of the original data. However, because the original data are not fully determined by the expression, it evaluates to a randomly chosen value of the appropriate type and size.

```
$ fun --m=double --c %f
conditional(
   field &,
   couple(constant 0,field &),
   constant 0)
$ fun --m=double --c %fO
12%fOi&
$ fun --m="12%fOi&" --c %fO
12%fOi&
$ fun --m="12%fOi&" --c %f
race(distribute,member)
$ fun --m="12%fOi&" --c %f
refer map transpose
```

Note that in the last two cases, above, the expression `12%fOi&` is seen to have different values on different runs. This effect is a consequence of the randomness inherent in its semantics. (It's best not to expect anything too profound from a randomly generated function.)

Inexact sizes Some primitive types are limited to particular sizes that can't be varied to order, such as booleans and floating point numbers. In such cases, the expression evaluates to an instance of the correct type at whatever size is possible.

```
$ fun --m="100%eOi&" --c %eO
62%eOi&
```

Opaque characters Opaque data expressions will usually be evaluated differently for every run, but an exception is made for opaque characters. In this case, the number ⟨*size*⟩ appearing in the expression is not the size of the data (which would always be in the range of 3 through 7 quits for a character), but the ISO code of the character. It uniquely identifies the character and will be evaluated accordingly.

```
$ fun --m="65%cOi&" --c %c
`A
$ fun --m="65%cOi&" --c %c
`A
```

However, a random character can be generated either by a size parameter in excess of 255 or an operand other than `&`, or both.

```
$ fun --m="256%cOi&" --c %c
229%cOi&
$ fun --m="65%cOi(0)" --c %c
175%cOi&
```

`Q` – Compressed

Any type expression ending with `Q` represents a compressed form of the type preceding the `Q`. For example, the type `%sLQ` is that of compressed lists of character strings. The compressed data format involves factoring out common subexpressions at the level of the virtual machine code representation.

- The compression is always lossless.
- It can take a noticeable amount of time for large data structures or functions.
- Compression rarely saves any real memory on short lived run time data structures, because the virtual machine transparently combines shared data when created by copying or detected by comparison.
- Compression saves considerable memory (possibly orders of magnitude) for redundant data that have to be written to binary files and read back again, because information about transparent run time sharing is lost when the data are written.

Listing 3.3 a list of non-unique character strings is a candidate for compression

```
long = # redundant data due to a repeated line

-[resistance is futile
you will be compressed
you will be compressed]-

short = # compressed version of the above data

%Q long
```

Compression function The way to construct an instance of a compressed type `%`t`Q` from an instance x of the ordinary type `%`t is by applying the function `%Q` to x. The function `%Q` takes an argument of any type and compresses it where possible. Note that `%Q` by itself is not a type expression but a function.

Extraction function Extraction of compressed data can be accomplished by the function `%QI`. This function takes any result previously returned by `%Q` and restores it to its original form, except in the degenerate case of `%Q 0`.

The `%QI` function can also be used as a predicate to test whether its argument represents compressed data. It will return an empty value if it does not, and return a non-empty value otherwise (normally the uncompressed data). However, to be consistent with this interpretation, `%QI %Q 0` evaluates to `&` (true) rather than `0`.[7]

Demonstration Not all data are able to benefit from compression, because it depends on the data having some redundancy. However, lists of non-unique character strings are suitable candidates. Given a source file `borg.fun` containing the text shown in Listing 3.3, we can see the effect of compression by executing a command to display the data in opaque format with and without compression.

```
$ fun borg.fun --main="(long,short)" --c %ooX
(504%oi&,338%oi&)
```

The output shows that the latter expression requires fewer quits for its encoding. If the above example is not sufficiently demonstrative, the effect can also be exhibited by the raw data.

```
$ fun borg.fun --m="(long,short)" --c %xW
(
   -{
      {{m[{cu[t@[mZSjCxbxS\H[qCxbtTS^d[qCtUz?=zF]zDAwH
      S\l[^[\>Ohm[^Wgz<EJ>Svd[gzFCtdbvd[^mjDStdbvB[^]z
```

[7] The alternative would be to use a function like `-+&&~& ~=&,%QI+-` for decompression if compressed empty data are a possibility, or the `extract` function from the `ext.avm` library distributed with the compiler.

```
      DSt>At^S^]zezf[^EZ`AtNCvezJ[I=Z@]z>mTB[i=Z<b=CtB
      [eJCl@[f=]w]x<@TBCe\M\E\<}-,
   -{
      zkKzSzPSauEkcyMz=CtfCw]z?=z<mzoAtTS\>O]cv{^=ZfCt
      ctdbzEjDStE[^]zFCt^S^mjf[dUz@]z<]ZpAvctB[e=Z=Ctu
      xt[<hR=]t>T@VNV\<}-)
```

Compressed data can be extracted automatically for printing as shown.

```
$ fun borg.fun --main=short --c %sLQ
%Q <
   'resistance is futile',
   'you will be compressed',
   'you will be compressed'>
```

where the output includes `%Q` as a reminder that the data were compressed, and to ensure that the data would be compressed again if the output were compiled. Decompression can also be performed explicitly by `%QI`, whereupon the result is no longer a compressed type.

```
$ fun borg.fun --main="%QI short" --c %sL
<
   'resistance is futile',
   'you will be compressed',
   'you will be compressed'>
```

`s` – Set

Analogously to the notation used for lists, a finite set can be expressed by a comma separated sequence of its elements enclosed in braces. The elements of a set can be of any type, including functions, although it is customary to think of all elements of a given set has having the same type, even if that type is a free union. The base type t in a set type expression `%`t`S` is the type of the elements.

Contrary to the practice with lists, the order in which the elements of a set are written down is considered irrelevant, and repetitions are not significant. Sets are therefore represented as lists sorted by an arbitrary but fixed lexical relation, followed by elimination of duplicates. These operations are performed transparently by the compiler at the time the expression in braces is evaluated.

```
$ fun --m="{'a','b'}" --c %sS
{'a','b'}
$ fun --m="{'b','a'}" --c %sS
{'a','b'}
$ fun --m="{'a','b','a'}" --c %sS
{'a','b'}
```

Because sets and lists have similar concrete representations, many list operations such as mapping and filtering are applicable to sets, using the same code. However, it is the user's responsibility to ensure that the transformation preserves the invariants of lexical ordering and no repetitions in the concrete representation of a set. One safe way of doing so is to compose list operations with the list-to-set pointer `~&s`, documented in the previous chapter on page 65.

`T` – Tree

The `T` type constructor is appropriate for trees in which each node can have arbitrarily many descendents, and all nodes have the same type. The base type t in a type expression `%`t`T` is the type of the nodes in the tree. This type constructor is a unary form of the dual type tree type constructor, `D`, explained on page 122. A type expression `%`t`T` is equivalent to `%`tt`D`.

Tree syntax An instance of a tree type `%`t`T` is expressed in the syntax

$$\langle \textit{root} \rangle \texttt{\^{}:}\ \texttt{<}[\langle \textit{subtree} \rangle[\texttt{,}\langle \textit{subtree} \rangle]^*]\texttt{>}$$

with the root having type `%`t. Each subtree is either an expression of the same form, or the empty tree, `~&V()`. For a tree with no descendents, the syntax is

$$\langle \textit{root} \rangle \texttt{\^{}:}\ \ \texttt{<>}$$

In either case above, the space after the `^:` operator is optional, but the lack of space before it is required. An alternative to this syntax sometimes used for printing is

$$\texttt{\^{}:}\ \ \texttt{(}\langle \textit{root} \rangle\ \texttt{,<}[\langle \textit{subtree} \rangle[\texttt{,}\langle \textit{subtree} \rangle]^*]\texttt{>)}$$

In the usage above, the space after the `^:` operator is required. It is also equivalent to write

$$\texttt{\^{}:<}[\langle \textit{subtree} \rangle[\texttt{,}\langle \textit{subtree} \rangle]^*]\texttt{>}\quad \langle \textit{root} \rangle$$

In this usage, the absence of a space after the `^:` operator is required, and the space between the subtrees and the root is also required. (Conventions regarding white space with operators are explained and motivated further in Chapter 5.)

Example As a small example, an instance of tree of `mpfr` (arbitrary precision) numbers, with type `%ET`, can be expressed in this syntax as shown.

```
-8.820510E+00^: <
   -1.426265E-01^: <
      ^: (
         -6.178860E+00,
         <3.562841E+00^: <>,6.094301E+00^: <>>)>,
   5.382370E+00^: <>>
```

W – Pair

The W type constructor is a unary type constructor describing pairs in which both sides have the same type. A type expression %tW is equivalent to %ttX. (The binary type constructor X is explained on page 123.) The same concrete syntax applies, which is that a pair is written ($\langle left \rangle$,$\langle right \rangle$), with $\langle left \rangle$ and $\langle right \rangle$ formatted according to the syntax of the base type.

An example of a type expression using this constructor is %nW, for pairs of natural numbers, and an instance of this type could be expressed as (120518122164,35510938).

Z – Maybe

The Z type constructor with a base type %t specifies a type that includes all instances of %t, with the same concrete representation and the same syntax, and also includes an empty instance. The empty instance could be written as () or [], depending on the base type.

```
$ fun --m="(1,2)" --c %nW
(1,2)
$ fun --m="(1,2)" --c %nWZ
(1,2)
$ fun --m="()" --c %nW
fun: writing `core'
warning: can't display as indicated type; core dumped
$ fun --m="()" --c %nWZ
()
```

The core dump in such cases is a small binary file containing a diagnostic message and the requested expression written in raw data (%x) format.

The usual applications for a maybe type are as an optional field in a record, an optional parameter to a function, or the result of a partial function when it's meant to be undefined. Although floating point numbers of type %e and %E have distinct maybe types %eZ and %EZ, it is probably more convenient to use NaN for undefined numerical function results, which propagates automatically through subsequent calculations according to IEEE standards, and does not cause an exception to be raised.

Some primitive types, such as %b, %g, %n, %s, %t, and %x, already have an empty instance, so they are their own maybe types. Any types constructed by D, G, L, N, S, T, and Z also have an empty instance already, so they are not altered by the Z type constructor.

The types for which Z makes a difference are %a, %c, %e, %f, %j, %q, %y, and %E, any record type, and anything constructed by A, J, Q, W. or X. For union types, both subtypes have to be one of these in order for the Z to have any effect.

m – Module

The m type constructor in a type %tm is mnemonic for "module". A module of any type %t is semantically equivalent to a list of assignments of strings to that type, %stAL, and the syntax is consistent with this equivalence. An example of a module of natural numbers, with type `%nm`, is the following.

```
<
   'foo': 42344,
   'bar': 799191,
   'baz': 112586>
```

Modules are useful in any kind of computation requiring small lookup tables, finite maps, or symbol environments.

- Modules can be manipulated by ordinary list operations, such as mapping and filtering.
- The dash operator allows compile time constants in modules to be used by name like identifiers. For example, if `x` were declared as the module shown above, then `x-foo` would evaluate to `42344`.
- The `#import` directive can be used to include any given module into the compiler's symbol table at compile time, in effect "bulk declaring" any computable list of values and identifiers.[8]

Usage of operators and directives is explained more thoroughly in subsequent chapters.

3.3 Remarks

There is more to learn about type expressions than this chapter covers, but readers who have gotten through it deserve a break, so it is worth pausing here to survey the situation.

- All primitive types and all but three idiosyncratic type constructors supported by the language are now at your disposal.
- While perhaps not yet in a position to write complete applications, you have substantially mastered much of the syntax of the language by learning the syntax for primitive and aggregate types explained in this chapter.
- The perception of different types as alternative descriptions of the same underlying raw data will probably have been internalized by now, along with the appreciation that they are all under your control.
- Your ability to use type expressions at this stage extends to
  - expressing parsers for selected primitive types
  - displaying expressions as the type of your choice using the `--cast` command line option
  - construction of compressed data and their extraction

[8] The compiler doesn't have a symbol table as such, but that's a matter for Part IV.

  - construction and extraction of data in self-describing format

- You've learned the meaning of the word "quit".

A sane society would either kill me or find a use for me.
Anthony Hopkins as Hannibal Lecter

4

Advanced usage of types

The presentation of type expressions is continued and concluded in this chapter, focusing specifically on several more issues.

- functions and exception handlers specified in whole or in part by type expressions, and their uses for debugging and verification of assertions
- abstract and self-modifying types via record declarations, and their relation to literal type expressions and pointer expressions
- a broader view of type expressions as operand stacks, with the requisite operators for data parameterized types and self-referential types

4.1 Type induced functions

Several ways of specifying functions in terms of type expressions are partly introduced in the previous chapter for motivational reasons, such as `p`, `Q`, `I`, `Y`, and `i`, but it is appropriate at this point to have a more systematic account of these operators and similar ones.

The relevant type expression mnemonics are shown in Table 4.1. These can be divided broadly between those that are concerned with exceptional conditions, useful mainly during development, and the remainder that might have applications in development and in production code. The latter are considered first because they are the easier group.

4.1.1 Ordinary functions

In this section, we consider type induced functions for printing, parsing, recognition, and the construction of self describing type instances, but first, one that's easier to understand than to motivate.

mnemonic	arity	meaning
`k`	1	identity function
`p`	1	parsing function
`C`	1	exceptional input printer
`I`	1	instance recognizer
`M`	1	error messenger
`P`	1	printer
`R`	1	recursifier (for `C` or `V`)
`Y`	1	self-describing formatter
`V`	2	i/o type validator

Table 4.1: one of these at the end of a type expression makes it a function

k – Identity function

The `k` type operator appended to any correctly formed type expression or type induced function transforms it to the identity function. It doesn't matter how complicated the function or type expression is.

```
$ fun --main="%cjXsjXDMk" --decompile
main = field &
$ fun --main="%nsSWnASASk" --decompile
main = field &
$ fun --main="%sLTLsLeLULXk" --decompile
main = field &
$ fun --main="%sLTLsLeLULXk -[hello world]-" --show
hello world
```

The application for this feature is to "comment out" type induced functions from a source text without deleting them entirely, because they may be useful as documentation or for future development.[1]

- As a small illustration, one could envision a source text that originally contains the code fragment `foo+ bar`, where `foo` and `bar` are functions and + is the functional composition operator.
- In the course of debugging, it is changed to `foo+ %eLM+ bar` for diagnostic purposes, using the `M` type operator explained subsequently, to verify the output from `bar`.
- When the issue is resolved, the code is changed to `foo+ %eLMk+ bar` rather having the diagnostic function deleted, leaving it semantically equivalent to the original because the expression ending with `k` is now the identity function.

Without any extra effort by the developer, there is now a comment documenting the output type of `bar` and the input type of `foo` as a list of floating point numbers. The same

[1] or perhaps "`k`omment out"

effect could also have been achieved by `foo+ (#%eLM+#) bar` using comment delimiters, but the more cluttered appearance and extra keystrokes are a disincentive. The resulting code would be the same in either case, because identity functions are removed from compositions during code optimization.

`p` – Parsing function

The mnemonic `p` appended to certain primitive type expressions results in a parser for that type, as explained in Section 3.1.1. The applicable types are `%a`, `%c`, `%e`, `%E`, `%n`, `%q`, `%s`, and `%x`, as shown in Table 3.1.

The parsing function takes a list of character strings to an instance of the type, and is an inverse of the printing function explained subsequently in this section. The character strings in the argument to the parsing function are required to conform to the relevant syntax for the type.

`I` – Instance recognizer

For a type %t, the instance recognizer is expressed %tI. Given an argument x of any type, the function %tI returns a value of `0` if x is not an instance of the type %t, and a non-zero value otherwise. For example, the instance recognizer for natural numbers, `%nI`, works as follows.

```
$ fun --m="%nI 10000" --c %b
true
$ fun --m="%nI 1.0e4" --c %b
false
```

The determination is based on the virtual machine level representation of the argument, without regard for its concrete syntax. Some values are instances of more than one type, and will therefore satisfy multiple instance recognizers.

```
$ fun --m="%eI 1.0e4" --c %b
true
$ fun --m="%cLI 1.0e4" --c %b
true
```

All instance recognizer functions follow the same convention with regard to empty or non-empty results, making them suitable to be used as predicates in programs. However, for some types, the value returned in the non-empty case has a useful interpretation relevant to the type.

Compressed type recognizers The compressed type instance recognizer %tQI has to uncompress its argument to decide whether it is an instance of %t. If it is an instance, and it's not empty, then the uncompressed argument is returned as the result. If it's an instance but it's empty, then `&` is returned. See page 131 for further explanations.

Function recognizers If the argument to the function instance recognizer `%fI` can be interpreted as a function, it is returned in disassembled form as a tree of type `%sfOXT`. The right side of each node is the semantic function needed to reassemble it, and the left side is a virtual machine combinator mnemonic.

```
$ fun --m="%fI compose(transpose,cat)" --c %sfOXT
('compose',48%fOi&)^: <
   ('transpose',7%fOi&)^: <>,
   ('cat',5%fOi&)^: <>>
```

This form is an example of a method used generally in the language to represent terms over any algebra. The semantic function in each node follows the convention of mapping the list of values of the subtrees to the value of the whole tree. This feature makes it compatible with the `~&K6` pseudo-pointer explained on page 85, which therefore can be used to resassemble a tree in this form.

```
$ fun --m="~&K6 %fI compose(transpose,cat)" --decompile
main = compose(transpose,cat)
```

Other function recognizers The job type recognizer `%`tJI behaves similarly to the function recognizer. For an argument of the form `~&J(`f`,`a`)`, where a is of type t, the result returned will be a disassembled version of f, as above. The same is true of the recognizers `%fZI`, `%fOI`, `%fOZI`, *etcetera*. Recognizers of assignments and pairs whose right sides are functions will also return the disassembled function if recognized.

`P` – Printer

For any type expression `%`t, a printing function is given by `%`t`P`, which will take an instance of the type to a list of character strings. The output contains a display of the data in whatever concrete syntax is implied by the type expression.

```
$ fun --m="%nLP <1,2,3,4>" --cast %sL
<'<1,2,3,4>'>
$ fun --m="%tLLP <1,2,3,4>" --cast %sL
<'<<&>,<0,&>,<&,&>,<0,0,&>>'>
$ fun --m="%bLLP <1,2,3,4>" --cast %sL
<
   '<',
   '   <true>,',
   '   <false,true>,',
   '   <true,true>,',
   '   <false,false,true>>'>
```

Note that the output in every case is cast to a list of strings `%sL`, because printing functions return lists of strings regardless of their arguments or their argument types. On the other hand, the `--cast` option isn't necessary if the output is known to be a list of strings.

```
$ fun --m="%bLLP <1,2,3,4>" --show
<
   <true>,
   <false,true>,
   <true,true>,
   <false,false,true>>
```

A few other points are relevant to printing functions.

- In contrast with parsing functions, which work only on a small set of primitive types, printing functions work with any type expression.
- In contrast with the `--cast` command line option, printing functions don't check the validity of their argument. They will either raise an exception or print misleading results if the input is not a valid instance of the type to be printed.
- Being automatically generated by the compiler from its internal tables, printing functions for non-primitive types are not as compact as the equivalent hand written code would be, making them disadvantageous in production code.
- Printing functions for aggregate types probably shouldn't be used in production code for the further reason that end users shouldn't be required to understand the language syntax.

`Y` – Self-describing formatter

The self describing formatter, `Y`, when used in an expression of the form `%`t`Y`, is a function that takes an argument of type `%`t to a result of type `%y`, the self describing type. The result contains the original argument and the type tag derived from `%`t, as required by the concrete representation for values of type `%y`.

This operation is briefly recounted here in the interest of having the explanations of all type induced functions collected together in this section, but a thorough discussion in context with motivation and examples is to be found starting on page 120.

4.1.2 Exception handling functions

It's a sad fact that programs don't always run smoothly. Hardware glitches, network downtime, budget cuts, power failures, security breaches, regulatory intervention, BWI alerts, and segmentation faults all take their toll. Most of these phenomena are beyond the scope of this document. Programs in Ursala can never cause a segmentation fault, except through vulnerabilities introduced by external libraries written in other languages.[2] However, there is a form of ungraceful program termination within our remit.

When the virtual machine is unable to continue executing a program because it has called for an undefined operation, it terminates execution and reports a diagnostic message obtained either by interrogation of the program or by default. These events are preventable

[2]or by a bug in the virtual machine, of which there are none known and none discovered through several years of heavy use

in principle by better programming practice, and considered crashes for the present discussion.

The supported mechanism for reporting of diagnostic messages during a crash is versatile enough to aid in debugging. Full details are documented in the `avram` reference manual, but in informal terms, it is a simple matter to supply a wrapper for any misbehaving function adding arbitrarily verbose content to its diagnostic messages. It is also possible to interrupt the flow of execution deliberately so as to report a diagnostic given by any computable function. Often the most helpful content is a display of an intermediate result in a syntax specified by a type expression. The functions described in this section take advantage of these opportunities.

`C` – Exceptional input printer

An expression of the form `%`t`C` denotes a second order function that can be used to find the cause of a crash. For a given function f, the function `%`t`C` f behaves identically to f during normal operation, but returns a more informative error message than f in the event of a crash.

- The content of the message is a display of the argument that was passed to f causing it to crash, followed by the message reported by f, if any.
- The original argument passed to f is reported, independent of any operations subsequently applied to it leading up to the crash.
- The argument is required to be an instance of the type `%`t, and will be formatted according to the associated concrete syntax.
- If the display of the argument takes more than one line, it is separated from the original message returned by f by a line of dashes for clarity.

The expression `%C` by itself is equivalent to `%gC`, which causes the argument to be reported in general type format. This format is suitable only for small arguments of simple types.

Intended usage The best use for this feature is with functions that fail intermittently for unknown reasons after running for a while with a large dataset, but reveal no obvious bugs when tried on small test cases. Typically the suspect function is deeply nested inside some larger program, where it would be otherwise difficult to infer from the program input the exact argument that crashed the inner function. More tips:

- If the program is so large and the bug so baffling that it's impossible to guess which function to examine, the type operator with a numerical suffix (e.g., `%0`, `%1`, `%2` ...) can be used just like a crashing argument printer `%`t`C`, but with no type expression t required. The diagnostic will consist only of the literal number in the suffix. Start by putting one of these in front of every function (with different numbers) and the next run will narrow it down.

Listing 4.1 toy demonstration of the crasher type operator, `C`

```
#import std
#import nat

f = # takes predecessors of a list of naturals, but has a bug

map %nC predecessor         # this should get to the bottom of it

t = (%nLC f) <25,12,5,1,0,6,3>
```

- In particularly time consuming cases or when the input type is unknown, the usage of `%xC` will serve to capture the argument in binary format for further analysis. The output in raw data syntax can be pasted into the source text, or saved to a binary file with minor editing (see page 119).
- Very verbose diagnostic messages can be saved to a file by piping the standard error stream to it. The `bash` syntax is `$ myprog 2> errlog`, where `myprog` is any executable program or script, including the compiler.
- Judicious use of opaque types, especially for arguments containing functions, can reduce unhelpful output.

Unintended usage This feature is *not* helpful in cases where the cause of the error is a badly typed argument, because the type of the argument has to be known, at least approximately (unless one uses `%xC` and intends to figure out the type later). The `V` type operator explained subsequently in this section is more appropriate for that situation. An attempt to report an argument of the wrong type will either show incorrect results or cause a further exception.

Example Listing 4.1 provides a compelling example of this feature in an application of great sophistication and subtlety. The function `f` is supposed to take a list of natural numbers as input, and return a list containing the predecessor of each item. The `predecessor` function is undefined for an input of zero, and raises an exception with the diagnostic message of `natural out of range`. This case slipped past the testing team and didn't occur until the dataset shown in the listing was encountered in real world deployment. The dataset is too large for the problem to be found by inspection, so the code is annotated to elucidate it.

```
$ fun crsh.fun --c %nL
fun:crsh.fun:9:13: <25,12,5,1,0,6,3>
------------------------------------------------------------
0
------------------------------------------------------------
natural out of range
```

The output from the compilation shows two arguments displayed, because there are two nested crashing argument printers in the listing. The outer one, `%nLC`, pertains the whole function `f`, and properly shows its argument as a list of natural numbers, while the inner one is specific to the `predecessor` function and displays only a single number. The first four arguments to the `predecessor` function in the list were processed without incident and not shown, but the zero argument, which caused the crash, is shown.

- Generally only the innermost crashing argument printer that isolates the problem is needed, but they can always be nested where helpful.
- The line and column numbers displayed in the compiler's output refer only to the position in the file of the top level function application operator that caused the error, rarely the site of the real bug.
- When the bug is fixed, the crashing argument printers should be changed to `%nCk` and `%nLCk` instead of being deleted, especially if the correct types are hard to remember.

`M` – Error messenger

Whereas the `C` type operator adds more diagnostic information to a function that's already crashing, the `M` type operator instigates a crash. This feature is useful because sometimes a program can be incorrect without crashing, but its intermediate results can still be open to inspection. Often an effective debugging technique combines the two by first identifying an input that causes a crash with the `C` operator, and then stepping through every subprogram of the crashing program individually using the `M` operator.

Usage The evaluation of an expression of the form `%`t`M` x causes x to be displayed immediately in a diagnostic message, with the syntax given by the type `%`t. However, rather than applying an error messenger directly to an argument, a more common use is to compose it with some other function to confirm its input or output.

- If a function f is changed to `%`t`M;` f, the original f will never be executed, but a display will be reported of the argument it would have had the first time control reached it (assuming the argument is an instance of `%`t).
- If the function is changed to `%`u`M+` f, it will not be prevented from executing, and if it is reached, its output will be reported immediately thereafter, with further computations prevented.
- Another variation is to write `%`t`C` `%`u`M+` f, which will show both the input and the output in the same diagnostic, separated by a line of dashes. Note the absence of a composition operator after `C`, and the presence of one after `M`.
- For very difficult applications, it is sometimes justified to verify the code step by step, changing every fragment f`+` g`+` h to `%`t`M+` f`+` `%`u`Mk+` g`+` `%`v`Mk+` h, and commenting out each previous error messenger to test the next one. The result is that the code is more trustworthy and better documented.

Diagnosing type errors A catch-22 situation could arise when an error messenger is used to debug a function returning a result of the wrong type. In order for an error messenger to report the result, its type must be specified in the expression, but in order for the type of result to be discovered, it must be reported as such.

A useful technique in this situation is to specify successive approximations to the type on each execution. The first attempt at debugging a function `f` has `%oM+ f` in the source, to confirm at least that `f` is being reached. If `f` should have returned a pair of something, the size reported for the opaque data should be greater than zero.

The next step is to narrow down the components of the result that are incorrectly typed. If the type should have been `%`*ab*`X`, then error messengers of `%`*a*`oXM`, `%o`*b*`XM`, and `%ooXM` can be tried separately. However, it would save time to use free unions with opaque types, as in an error messenger of `%`*a*`oU`*b*`oUXM`. The incorrectly typed component(s) will then be reported in opaque format, while the correctly typed component, if any, will be reported in its usual syntax.

The technique can be applied to other aggregate types such as trees and lists, using an error messenger like `%`*a*`oUTM` or `%`*a*`oULM`. If only one particular node or item of the result is badly typed, then only that one will be reported in opaque format. In the case of record types (documented subsequently in this chapter) union with the opaque type in an error messenger will allow either the whole record or only particular fields to be displayed in opaque format, making the output as informative as possible.

`R` – Recursifier

The `R` type operator can be appended to expressions of the form `%`*t*`C` or `%`*t*`V`, to make them more suitable for recursively defined functions. If a recursive function f crashes in an expression of the form `%`*t*`CR` f, the diagnostic will show not just the argument to f, but the specific argument to every recursive invocation of f down to the one that caused the crash. The effect for `%`*t*`VR` f is analogous. The printer and verifier functions behave as documented in all other respects.

- The compiler will complain if `R` is appended to a type expression that doesn't end with `C` or `V`.
- The compiler will complain if this operation is applied to something other than a recursively defined function. A recursively defined function is anything whose root combinator in virtual code is `refer` (as shown by `--decompile`), which includes code generated by the `o` pseudo-pointer and several functional combining forms such as `*^` (tree traversal), `^&` (recursive conjunction), and `^?` (recursive conditional).

Example A certain school of thought argues against defensive programming on the basis that it's more manageable for a subprogram in a large system to crash than to exceed its documented interface specification when it's undefined. Listing 4.2 shows a tree traversing function `f` that doesn't work for empty trees by design. It also doesn't work for any tree with an empty subtree. Otherwise, for a tree of natural numbers, it doubles the number in

Listing 4.2 value of `f` is undefined for empty trees

```
#library+

x =         # random test data of type %nT

7197774595263^: <
   10348909689347579265^: <
      158319260416525061728777^: <
         0^: <>,
         ~&V(),
         574179086^: <
            ^: (
               1460,
               <0^: <>,1^: <>,1707091^: <>,30^: <>>)>>,
      213568^: <>,
      128636^: <97630998857^: <>>>>

f = ~&diNiCBPvV*^
```

every node by inserting a 0 in the least significant bit position. The listing is assumed to be in a source file named `rcrsh.fun`.

```
$ fun rcrsh.fun
fun: writing 'rcrsh.avm'
$ fun rcrsh --main=f --decompile
main = refer compose(
   couple(
      conditional(
         field(&,0),
         couple(constant 0,field(&,0)),
         constant 0),
      field(0,&)),
   couple(field(0,(&,0)),mapcur((&,0),(0,(0,&)))))
```

Let's find out what happens when the function `f` is applied to the test data `x` shown in the listing, which has an empty subtree.

```
$ fun rcrsh --main="f x" --c %nT
fun:command-line: invalid deconstruction
```

This is all as it should be, unless of course the function crashed for some other reason. To verify the chain of events leading to the crash, we can execute

```
$ fun rcrsh --main="(%nTCR f) x" --c %nT 2> errlog
```

and view the crash dump file `errlog` (or whatever name was chosen) whose contents are reproduced in Listing 4.3. Alternatively, a more concise crash dump is obtained by using opaque types.

Listing 4.3 recursive crash dump from Listing 4.2 showing the chain of calls leading to a crash

```
fun:command-line: 7197774595263^: <
   10348909689347579265^: <
      158319260416525061728777^: <
         0^: <>,
         ~&V(),
         574179086^: <
            ^: (
               1460,
               <0^: <>,1^: <>,1707091^: <>,30^: <>>)>>,
      213568^: <>,
      128636^: <97630998857^: <>>>>
----------------------------------------------------------------------
10348909689347579265^: <
   158319260416525061728777^: <
      0^: <>,
      ~&V(),
      574179086^: <
         ^: (
            1460,
            <0^: <>,1^: <>,1707091^: <>,30^: <>>)>>,
   213568^: <>,
   128636^: <97630998857^: <>>>
----------------------------------------------------------------------
158319260416525061728777^: <
   0^: <>,
   ~&V(),
   574179086^: <
      ^: (
         1460,
         <0^: <>,1^: <>,1707091^: <>,30^: <>>)>>
----------------------------------------------------------------------
~&V()
----------------------------------------------------------------------
invalid deconstruction
```

```
$ fun rcrsh --main="(%oCR f) x"
fun:command-line: 499%oi&
------------------------------------------------------------
430%oi&
------------------------------------------------------------
222%oi&
------------------------------------------------------------
0%oi&
------------------------------------------------------------
invalid deconstruction
```

The zero size of the last argument means it can only be empty, which demonstrates that the crash was caused specifically by an empty subtree. Of course, it also would be necessary in practice to verify that the function doesn't crash and gives correct results for valid input, but this issue is beyond the scope of this example.

V – Type validator

For a given function f, an expression of the form `%`ab`V` f represents a function that is equivalent to f whenever the input to f is an instance of type `%`a and the output from f is of type `%`b, but that raises an exception otherwise.

- If the input to a function of the form `%`ab`V` f is not an instance of the type `%`a, the diagnostic message reported when the exception is raised will be the words "`bad input type`". The function f is not executed in this case.
- If the input is an instance of `%`a, the function f is applied to it. If the output from f is not an instance of `%`b, the diagnostic message will report the input in the concrete syntax associated with `%`a, followed by a line of dashes, followed by the words "`bad output type`".
- If f itself causes an exception in the second case, only the diagnostic from f is reported.

The type operator `V` is best understood as a binary operator in that it requires two subexpressions in the type expression where it occurs, a and b. Its result is not a type expression but a second order function, which takes a function f as an argument and returns a modified version of f as a result. The modified version behaves identically to f in cases of correctly typed input and output. [3]

Validator usage This feature is useful during development for easily localizing the origin of errors due to incorrect typing. It might also be useful during beta testing but probably not in production code, due to degraded performance, increased code size, and user unfriendliness.

[3] Advocates of strong typing may see this section as a vindication of their position. It's true that you don't have these problems with a strongly typed language (or at least not after you get it to compile), but on the other hand, you aren't allowed to write most applications in the first place.

Although the type validation operator pertains to both the input and the output types of a function, it would be easy to code a validator pertaining to just one of them by using a type that includes everything for the other.

- If a function is polymorphic in its input but has only one type of output (for example, a function that computes the length of list of anything), it is appropriate to use a validator of the form `%o`*t*`V` or `%x`*t*`V` on it, which will concern only the output type. The latter will be more helpful for finding the cause of a type error, if any, by reporting the input that caused the error in raw format.
- A validator like `%`*t*`xV` is meaningful in the case of a function with only one input type but many output types (for example, a function that extracts the data field from self-describing `%y` type instances).
- This technique can be extended to functions with more limited polymorphism by using free unions. For example, `%ejUjV` would be appropriate for a function that takes either a real or a complex argument to a complex result.
- Some useless validators are `%xxV` and `%ooV`, which have no effect.

Example A naive implementation of a function to perform a bitwise AND operation on a pair of natural numbers is given by the following pseudo-pointer expression.

```
$ fun --main="~&alrBPalhPrhPBPfabt2RCNq" --decompile
main = refer conditional(
   conditional(field(0,(&,0)),field(0,(0,&)),constant 0),
   couple(
      conditional(
         field(0,((&,0),0)),
         field(0,(0,(&,0))),
         constant 0),
      recur((&,0),(0,(((0,&),0),(0,(0,&)))))),
   constant 0)
```

The problem with this function is that the result is not necessarily a valid representation of a natural number, because it doesn't maintain the invariant that the most significant bit should be `&`.

This error can be detected through type validation with sufficient testing. In practice we might run the program on a large randomly generated test data set, but for expository purposes a couple of examples are tried by hand. On the first try, it appears to be correct.

```
$ fun --m="(%nWnV ~&alrBPalhPrhPBPfabt2RCNq) (8,24)" --c
8
```

On the second try, the invalid output is detected.

```
$ fun --m="(%nWnV ~&alrBPalhPrhPBPfabt2RCNq) (8,16)" --c
fun:command-line: (8,16)
------------------------------------------------------------
bad output type
```

Because the function is recursively defined, we can also try the `R` operator on it for more information.

```
$ fun --m="(%nWnVR ~&alrBPalhPrhPBPfabt2RCNq) (8,16)" --c
fun:command-line: (8,16)
------------------------------------------------------------
(4,8)
------------------------------------------------------------
(2,4)
------------------------------------------------------------
(1,2)
------------------------------------------------------------
bad output type
```

This result shows that even an input as simple as `(1,2)` would cause a type error. To get a better idea of the problem, we examine the raw data.

```
$ fun --m="~&alrBPalhPrhPBPfabt2RCNq (1,2)" --c %tL
<0>
```

This result combined with a mental simulation of the listing of the decompiled virtual code above is enough to identify the problem.

4.2 Record declarations

Difficult programming problems are made more manageable by the time honored techniques of abstract data types. The object oriented paradigm takes this practice further, with a tightly coupled relationship between code and data, and interfaces whose boundaries are carefully drawn. The functional paradigm promotes an equal footing for functions and data, largely subsuming the characteristics of objects within traditional records or structures, because their fields can be functions. However, one benefit of objects remains, which is their ability to be initialized automatically upon creation and to maintain specified invariants automatically during their existence.

The present approach draws on the strengths of object orientation to the extent they are meaningful and useful within an untyped functional context. The mechanism for abstract data types is called a record in this manual, and it plays a similar rôle to records or structures in other languages. The terminology of objects is avoided, because methods are not distinguished from data fields, which can contain functions. However, an additional function can be associated optionally with each field, which initializes or updates it implicitly whenever its dependences are updated. These features are documented in this section.

Listing 4.4 a library exporting an untyped record with three fields and an example instance

```
#library+

myrec :: front middle back

an_instance = myrec[front: 2.5,middle: 'a',back: 1/3]
```

4.2.1 Untyped records

The simplest kind of record declaration is shown in Listing 4.4, which has a record named `myrec` with fields named `front`, `middle`, and `back`. A record declaration may be stored for future use in a library by the `#library+` directive, or used locally within the source where it is declared.

Field identifiers

If a record is declared by no more than the names of its fields, it serves as a user defined container for values of any type. In this regard, it is comparable to a tuple whose components are addressed by symbolic names rather than deconstructors like `&l` and `&r`. In fact, the field identifiers are only symbolic names for addresses chosen automatically by the compiler, and can be treated as data. With Listing 4.4 in a file named `rlib.fun`, we can verify this fact as shown.

```
$ fun rlib.fun
$ fun: writing `rlib.avm'
$ fun rlib --main="<front,middle,back>" --cast %aL
<2:0,2:1,1:1>
```

Record mnemonics

The record mnemonic appears to the left of the double colons in a record declaration, and has a functional semantics.

- If the record mnemonic is applied to an empty argument, it returns an instance of the record in which all fields are addressable (i.e., without causing an invalid deconstruction exception) but empty.
- If the record mnemonic is applied to a non-empty argument, the argument is treated as a partially specified instance of the record, and the function given by the mnemonic fills in the remaining fields with empty values or their default values, if any.

For an untyped record such as the one in Listing 4.4, the empty form and the initialized form of the record are the same, because the default value of each field is empty. In general, the empty form provides a systematic way for user defined polymorphic functions to ascertain the number of fields and their memory map for a record of any type.[4]

For the example in Listing 4.4, the record mnemonic is `myrec`, and has the following semantics.

```
$ fun rlib --m=myrec --decompile
main = conditional(
   field &,
   couple(
      compose(
         conditional(field &,field &,constant &),
         field(&,0)),
      field(0,&)),
   constant 1)
```

This function would be generated for the mnemonic of any untyped record with three fields, and will ensure that each of the three is addressable even if empty.

```
$ fun rlib --m="myrec ()" --c %hhZW
(((),()),())
```

However, the main reason for using a record is to avoid having to think about its concrete representation, so neither the record mnemonic nor the default instance would ever need to be examined to this extent.

Instances

An instance of a record is normally expressed by a comma separated sequence of assignments of field identifiers to values, enclosed in square brackets, and preceded by the record mnemonic.

$$
\begin{array}{l}
\langle \textit{record mnemonic} \rangle\, \texttt{[} \\
\qquad\qquad \langle \textit{field identifier} \rangle \texttt{:} \quad \langle \textit{value} \rangle\texttt{,} \\
\qquad\qquad\qquad\qquad\quad \vdots \\
\qquad\qquad \langle \textit{field identifier} \rangle \texttt{:} \quad \langle \textit{value} \rangle\texttt{]}
\end{array}
$$

The fields can be listed in any order, and can be omitted if their default values are intended. The code in Listing 4.4 would have worked the same if the declaration of the instance had been like this.

```
an_instance = myrec[back: 1/3,front: 2.5,middle: 'a']
```

To initialize only the `middle` field and leave the others to their default values, the syntax would be like this.

[4] There is of course no concept of mutable storage in the language. References to updating and initialization throughout this manual should be read as evaluating a function that returns an updated copy of an argument. For those who find a description is these terms helpful, all arguments to functions are effectively "passed by value". Although the virtual machine is making pointer spaghetti behind the scenes, sharing is invisible at the source level.

```
an_instance = myrec[middle: 'a']
```

The record mnemonic is necessary to supply any implicit defaults. This syntax is similar to that of an a-tree (page 128), except that the addresses are symbolic rather than literal. Unlike lists, sets, and a-trees, there is no expectation that all fields in a record should have same type.

In some situations, it is convenient to initialize the values of a pair of fields by a function returning a pair, so a variation on the above syntax can be used as exemplified below.

```
point[(y,x): mpfr..sin_cos 1.2E0, floating: true]
```

The `mpfr..sin_cos` function used in this example computes a pair of numbers more efficiently than computing each of them separately.

To express an instance of a record in which all fields have their default values, a useful idiom is ⟨*record mnemonic*⟩`&`. That is, the record mnemonic is applied to the smallest non-empty value, `&`.

Deconstruction

The field identifiers declared with a record can be used as deconstructors on the instances.

```
$ fun rlib --m="~front an_instance" --c %e
2.500000e+00
$ fun rlib --m="~middle an_instance" --c %s
'a'
$ fun rlib --m="~back an_instance" --c %q
1/3
$ fun rlib --m="~(front,back) an_instance" --c %eqX
(2.500000e+00,1/3)
```

The values that are extracted are consistent with those that are stored in the record instance shown in Listing 4.4. The dot operator is a useful way of combining symbolic with literal pointer expressions.

```
$ fun rlib --m="~middle.&h an_instance" --c %c
`a
```

An expression of the form `~`a`.`$b\ \ x$ is equivalent to `~`b `~`$a\ \ x$, except where a is a pointer with multiple branches, in which case it follows the rules discussed in connection with the composition pseudo-pointer (page 79). To ensure correct disambiguation, this usage of the dot operator permits no adjacent spaces.

Implicit type declarations

Whenever a record is declared by the `::` operator, a type expression is implicitly declared as well, whose identifier is the record mnemonic preceded by an underscore. Identifiers with leading underscores are reserved for implicit declarations so as not to clash with user

Listing 4.5 Typed records annotate some or all of the fields with a type expression.

```
#import std

#library+

goody_bag ::          # record declaration with typed fields

number_of_items  %n  # field types are specified like this
cost             %e
celebrity_rank   %cZ
occasion         %s
hypoallergenic   %b

goodies =             # an instance of the typed record

goody_bag[
   number_of_items: 6,
   cost:            125.00,
   celebrity_rank:  `B,
   occasion:        'Academy Awards',
   hypoallergenic:  true]
```

defined identifiers. The record type identifier can be used like any other type expression for casting or for type induced functions.

```
$ fun rlib --main=an_instance --cast _myrec
myrec[front: 57%oi&,middle: 6%oi&,back: 8%oi&]
```

Values cast to untyped records are printed with all fields in opaque format because there is no information available about the types of the fields, and with any empty fields suppressed. The opaque format nevertheless gives an indication of the sizes of the fields. The next example demonstrates a record instance recognizer.

```
$ fun rlib --main="_myrec%I an_instance" --cast %b
true
```

When a type expression given by a symbolic name is used in conjunction with other type constructors or functionals such as `I` and `P`, the symbolic name appears on the left side of the `%` in the type expression, and the literals appear on the right, as in t%u. This convention is a matter of necessity to avoid conflation of the two.

4.2.2 Typed records

The next alternative to an untyped record is a typed record, which is declared with the syntax exemplified in Listing 4.5.

- Typed records have an optional type expression associated with each field in the declaration.

- The type expression, if any, follows the field identifier in the declaration, separated by white space, with no other punctuation or line breaks required.
- There is usually no ambiguity in this syntax because type expressions are readily distinguishable from field identifiers, but the type expression optionally can be parenthesized, as in `(%cZ)`.
- Parentheses are necessary only when the type expression is given by a single user defined identifier without a leading underscore.

Typed record instances

The syntax for typed record instances is the same as that of untyped records, but there is an assumption that the field values are instances of their respective types. This assumption allows the record instance to be displayed with a more informative concrete syntax than the opaque format used for untyped records. If the source code in Listing 4.5 resides in file named `bags.fun`, the record instance would be displayed as shown.

```
$ fun bags.fun
fun: writing `bags.avm'
$ fun bags --m=goodies --c _goody_bag
goody_bag[
   number_of_items: 6,
   cost: 1.250000e+02,
   celebrity_rank: `B,
   occasion: 'Academy Awards',
   hypoallergenic: true]
```

Type checking

The instance checker of a typed record verifies not only that all fields are addressable, but that they are all instances of their respective declared types.

```
$ fun bags --m="_goody_bag%I 0" --c %b
false
$ fun bags --m="_goody_bag%I goody_bag[cost: 'free']" -c %b
false
$ fun bags --m="_goody_bag%I goody_bag[cost: 0.0]" --c %b
true
```

This convention applies also to the type validator operator, `V`, when used in conjunction with typed records (page 149), and to the `--cast` command line option, which will decline to display a badly typed record instance as such.

```
$ fun bags --m="goody_bag[cost: 'free']" --c _goody_bag
fun: writing `core'
warning: can't display as indicated type; core dumped
```

Listing 4.6 default values with nested records

```
t :: a %e b %q

u :: c _t d %E

#cast _u

x = u&    # default value of a record of type _u
```

Default values

Fields in a typed record sometimes have non-empty default values to which they are automatically initialized if left unspecified.

```
$ fun bags --m="goody_bag&" --c _goody_bag
goody_bag[cost: 0.000000e+00]
```

This example shows the default value of `0.0` automatically assigned to the `cost` field, even though no value was explicitly specified for it. These conventions are observed with regard to default values.

- If the empty value, `()`, is a valid instance of the field type, then that value is the default. Types with empty instances include naturals, strings, booleans, and all lists, sets, trees, grids, and "maybe" types (`%`*t*`Z`).
- Primitive types with non-empty default values include the numeric types `%e`, `%E`, and `%q`, whose defaults are `0.0`, `0.0E0`, and `0/1`. For the `%E` type, the minimum precision is used. The address type `%a` has a default value of `0:0`.
- If a field in a record is also a record, the default value of the field is given by the default value of the inner record.
- The default value of a record is the value obtained by initializing all of its fields to their default values.
- If a field in a record is a pair for which both sides have default values, the default value of the field is the pair of default values.

An example of a typed record with a field that is also a typed record is shown in Listing 4.6. When this code is compiled, the output is

```
u[c: t[a: 0.000000e+00,b: 0/1],d: 0.00E+00]
```

Some types, such as functions and characters, have neither an empty instance nor a sensible default value. If such a field is left unspecified, the record is badly typed. If there is sometimes a good reason for such a field to be undefined, then the corresponding "maybe" type should be used for that field in the record declaration.

Listing 4.7 Recursively defined records are a hundred percent legitimate.

```
contract :: main_clause %s subclauses _contract%L

hit =

contract[
   main_clause: 'yadayada',
   subclauses: <
      contract[main_clause: 'foo'],
      contract[
         main_clause: 'bar',
         subclauses: <
            contract[main_clause: 'lot'],
            contract[main_clause: 'of'],
            contract[main_clause: 'buffers']>],
      contract[main_clause: 'baz']>]
```

Recursive records

Typed records open the possibility of fields that are declared to be of record types themselves, by way of implicitly declared type identifiers as seen in previous examples, such as `_myrec` and `_goody_bag`. A hierarchy of record declarations used appropriately can be an important aspect of an elegant design style.

When multiple record declarations are used together, the issue inevitably arises of cyclic dependences among them. Circular definitions are generally not valid in Ursala except by special arrangement (i.e., with the `#fix` compiler directive), but in the case of record declarations, they are valid and are interpreted appropriately.[5]

Listing 4.7 briefly illustrates the use of recursion in a record declaration. In this case, only a single declaration is involved, and it depends on itself by invoking its own type identifier, `_contract`. Instances of this type can be cast or type checked as any other type. This technique is applicable in general to any number of mutually dependent declarations.

Although it serves to illustrate the idea of recursive records, the record in Listing 4.7 offers no particular advantage over the type of trees of strings, `%sT`. Trees are an inherently recursive container suitable for most applications in practice and are better integrated with other features of the language. However, one could undoubtedly envision some suitably complicated example for which only a user defined recursive container would suffice.

4.2.3 Smart records

The facility for automatically initialized fields in typed records can be taken a step further by having them initialized according to a specified function. Records with custom designed initialization functions are called smart records in this manual.

[5] only for the record declarations, not for mutually dependent declarations of instances of the records

Smart record syntax

The syntax for smart recard declarations is upward compatible with untyped records and typed records, consisting of a record mnemonic, followed by the record declaration operator `::`, followed by a white space separated sequence of triples of field identifiers, type expressions, and initializing functions.

$$
\begin{array}{l}
\langle \textit{record mnemonic} \rangle \ \texttt{::} \\
\quad \langle \textit{field identifier} \rangle \quad \langle \textit{type expression} \rangle \quad \langle \textit{initializing function} \rangle \\
\quad \vdots \\
\quad \langle \textit{field identifier} \rangle \quad \langle \textit{type expression} \rangle \quad \langle \textit{initializing function} \rangle
\end{array}
$$

Untyped and uninitialized fields may be mixed with initialized fields in the same declaration. For an initialized field, a type expression is required by the syntax, but an untyped initialized field can be specified either with an opaque type expression, `%o`, or an empty value `()` as a place holder. This syntax is usually unambiguous, but the initialization function can be parenthesized if necessary to distinguish it from a field identifier.

Semantics

The calling convention for the initializing function is that its argument is the whole record, and its result is the value of the field that it initializes. It will normally access any fields on which its result depends by deconstructor functions using their field identifiers in the normal way. An initializing function may raise an exception, which is useful if its purpose is only to verify an assertion or invariant.

A field in a record could be declared as a record type itself. In that case, the inner record is initialized first by its own initializing function before being accessible to the initializing functions of the outer record. The same applies to any type of field that has a non-empty default value.

If a field contains a list of records, every record in the list is first initialized locally before being accessible to the initializing functions at the outer level. The same applies to other containers, such as sets and a-trees, and other types having default values, such as floating point numbers.

If there are multiple fields with initializing functions in the same record, they are effectively evaluated concurrently. Any data dependences among them are resolved according to the following protocol.

- All field initializing functions are evaluated with identical inputs.
- When a result is obtained for every field, a new record is constructed from them.
- If any field in the new record differs from the corresponding field in the preceding one, the process is iterated.
- The result from any field initializing function is accessible by the others as of the next iteration.

Listing 4.8 polar and retangular coordinates automatically maintained

```
#import std
#import nat
#import flo

#library+

point :: # each field has a type and an initializer

x %eZ -|~x,-&~r,~t,times^/~r cos+ ~t&-,~r,! 0.|-
y %eZ -|~y,-&~r,~t,times^/~r sin+ ~t&-,! 0.|-
r %eZ -|~r,-&~x,~y,sqrt+ plus+ sqr^~/~x ~y&-,~x,~y,! 0.|-
t %eZ -|~t,-&~x,~y,math..atan2^/~y ~x&-,~y&& ! div\2. pi,! 0.|-

# functions

add    = point$[x: plus+ ~x~~,y: plus+ ~y~~]
rotate = point$[r: ~&r.r,t: plus+ ~/&l &r.t]
scale  = point$[r: times+ ~/&l &r.r,t: ~&r.t]
invert = scale/-1.
orbit  = scale/2.1+ add^/invert rotate/0.5
```

- Initialization terminates either when a fixed point is reached or a repeating cycle is detected.
- In the case of a cycle, the record instance with the minimum weight in the cycle is taken as the result, or with multiple minimum weights an arbitrary choice is made.

An initializing function never gets to see a record in which some fields have been initialized more than others. If multiple iterations are needed, every field will have been initialized the same number of times. In practical applications, very few iterations should be needed unless the initializing functions are inconsistent with one another. However, it is the user's responsibility to ensure convergence.

Example

Listing 4.8 shows a simple example of a smart record developed for a small library of operations on two dimensional real vectors or points in a plane. A point has two equivalent representations, either as a pair of cartesian cordinates (x, y), or as a pair of polar coordinates, (r, t), which are related as shown.

$$\begin{array}{ll} x = r\cos(t) & r = \sqrt{x^2 + y^2} \\ y = r\sin(t) & t = \arctan(y/x) \end{array}$$

The smart record allows a point to be specified either by its (x, y) coordinates or its (r, t) coordinates, and automatically infers the alternative. This feature is convenient because some operations are better suited to one representation than the other, and can be expressed in reference to the appropriate one. Moreover, compositions of different operations require no explicit conversions between representations.

Much of the code in Listing 4.8 involves language features introduced in subsequent chapters, so it is not discussed in detail at this stage. However, some crucial ideas should be noted.

- Addition uses the cartesian representation.
- Rotation and scaling use the polar representation.
- The orbit function composes four functions without reference to either representation and without explicit conversions.

To see smart records in action, we store Listing 4.8 in a file named `plib.fun` and compile it as follows.

```
$ fun flo plib.fun
fun: writing `plib.avm'
```

The remaining fields are initialized automatically when a value of `1.` is assigned to `y`.

```
$ fun plib --m="point[y: 1.]" --c _point
point[
   x: 0.000000e+00,
   y: 1.000000e+00,
   r: 1.000000e+00,
   t: 1.570796e+00]
```

The `scale` function changes only the r coordinate, but the others are automatically adjusted.

```
$ fun plib --m="scale/2. point[x: 0.5,y: 1.]" --c _point
point[
   x: 1.000000e+00,
   y: 2.000000e+00,
   r: 2.236068e+00,
   t: 1.107149e+00]
```

The same effect is achieved by adding a pair of equal points, even though only the x and y coordinates are directly referenced by the `add` function.

```
$ fun plib --m="add ~&iiX point[x: 0.5,y: 1.]" --c _point
point[
   x: 1.000000e+00,
   y: 2.000000e+00,
   r: 2.236068e+00,
   t: 1.107149e+00]
```

Listing 4.9 Parameterized records allow generic or polymorphic types.

```
#import std
#import nat

polyset "t" ::  # parameterized by the element type

elements    "t"%S
cardinality    %n  length+ ~elements

realset      = polyset %e
realset_type = _polyset %e

x = realset[elements: {1.0,2.0,3.0}]
y = (polyset %s)[elements: {'foo','bar'}]
```

4.2.4 Parameterized records

A way of defining general classes of records with a single declaration is to use a parameterized record, such as the one shown in Listing 4.9. The idea is that the common features of a class of records are fixed in the declaration, and the features that vary from one to another are represented by dummy variables.

- The dummy variables can be used in the declaration anywhere an identifier for a constant could be used, whether to parameterize the type expressions or the initializing functions. The same dummy variable can be used in several places.
- The record mnemonic has the semantics of a higher order function. When applied to a parameter value, the record mnemonic of a parameterized record instantiates the dummy variable as the parameter and returns a function that can be used as an ordinary record mnemonic.
- The implicitly declared type identifier of a parameterized record doesn't represent a type expression, but a function that takes a parameter as input and returns a type expression as a result. The result returned can be used like an ordinary type expression.

Applications

One application for parameterized records would be to specify a polymorphic type class. The parameter can determine the type of a field in the record, among other things. Another would be to implement optional or pluggable features in a field initializing function. However, there may be simpler solutions to these problems than parameterized records.

- Polymorphic records can be obtained in various ways by declaring the changeable fields as general, opaque, raw, or self-describing types (`%g`, `%o`, `%x`, or `%y`, respectively), or as a free union of some known set of types.

- If an initializing function requires a proliferation of optional configuration settings, the record can be declared with extra fields to store them. Every field in a record is accessible to every initialization function in it.

In fact, it is difficult to identify a compelling case for parameterized records. I (the author of the language) don't consider them a useful feature but have provided them partly as a friendly gesture to those who may feel otherwise, and partly as an exercise in compiler writing.

Syntax

For the simple case of a first order parameterized record, the syntax for the declaration is as follows.

$$\langle \textit{record mnemonic} \rangle \ \langle \textit{dummy variable} \rangle \ \mathbf{::} \ \langle \textit{fields} \rangle$$

- The ⟨*fields*⟩ have the syntax explained previously for typed or smart records, but may also employ free occurrences of dummy variables.
- The ⟨*dummy variable*⟩ can be a double quoted string containing any printable characters other than a double quote, and that is not broken across lines.
- Alternatively, lists and tuples of dummy variables are allowed in place of a single one, in any combination to any depth. They follow the usual syntax for lists and tuples in the language as comma separated sequences enclosed in angle brackets or parentheses.

Higher order parameterized records require one of the following forms, where the v's are dummy variables or lists or tuples thereof, as explained above.

$$\begin{array}{rcl} (\langle \textit{record mnemonic} \rangle \ v_0) \ v_1 & \mathbf{::} & \langle \textit{fields} \rangle \\ ((\langle \textit{record mnemonic} \rangle \ v_0) \ v_1) \ v_2 & \mathbf{::} & \langle \textit{fields} \rangle \\ (((\langle \textit{record mnemonic} \rangle \ v_0) \ v_1) \ v_2) \ v_3 & \mathbf{::} & \langle \textit{fields} \rangle \\ & \vdots & \end{array}$$

The parentheses in this usage are necessary and must be nested as shown to inhibit the usual right associativity of function application in the language. An alternative syntax for higher order records is the following.

$$\begin{array}{rcl} \langle \textit{record mnemonic} \rangle(v_0) \ v_1 & \mathbf{::} & \langle \textit{fields} \rangle \\ \langle \textit{record mnemonic} \rangle(v_0)(v_1) \ v_2 & \mathbf{::} & \langle \textit{fields} \rangle \\ \langle \textit{record mnemonic} \rangle(v_0)(v_1)(v_2) \ v_3 & \mathbf{::} & \langle \textit{fields} \rangle \\ & \vdots & \end{array}$$

In this form, the parentheses are optional but a lack of space before each dummy variable is compulsory, except before the last one. Juxtaposition without a space is interpreted as a left associative version of function application.

Usage

The use of a record mnemonic for a parameterized record must match its declaration, both in the order and the structure of the parameters. In this regard, it should be noted particularly by experienced functional programmers that there is a firm distinction in this language between a second order parameterized record and a first order record parameterized by a pair. That is,

```
(rec "a") "b" :: ...
```

is *not* semantically equivalent to

```
rec ("a","b") :: ...
```

Although they are similarly expressive, the latter has a somewhat more efficient implementation. The choice between them is a design decision, perhaps favoring the former when there is some reason to expect that `"a"` doesn't need to be changed as often as `"b"`.

First order If something is declared as a first order parameterized record `rec`, then a relevant record instance would be expressed as

```
(rec x)[...]
```

where `x` matches the size or arity of the parameter. That is, if `rec` were declared

```
rec ("a","b") :: ...
```

then the value of `x` should be a pair, so that its left side can be instantiated as `"a"` and its right side as `"b"`. If `rec` were declared as

```
rec <"u","v","w"> :: ...
```

then `x` should be a list of length three. If dummy variables occur in nested tuples or lists, the parameter should have a similar form.

Note that if `rec` is a parameterized record, then it is not correct to write `rec[...]` as a record instance without a parameter to the mnemonic, but it is possible to define a specific record type

```
some_rec = rec some_param
```

and then to express an instance as `some_rec[...]`.

Higher order If a higher order parameterized record is declared

```
(...((rec "a") "b")..."z") :: ...
```

the same considerations apply, with the additional provision that the nesting of function applications in the use of the mnemonic must match its declaration, and the innermost argument must match the structure of the innermost parameter. Hence, an instance of the relevant record would be expressed

```
(...((rec a_val) b_val)...z_val)[...]
```

Special cases of such a record can also be defined and invoked accordingly by fixing one or more of the inner parameters.

```
spec = rec a_val
```

An instance could then be expressed

```
(...(spec b_val)...z_val)[...]
```

Types The type identifier of a parameterized record follows the same calling conventions as the record mnemonic, but returns a type expression. Otherwise, all of the above discussion applies.

This situation is particularly relevant to recursively defined parameterized records, in which care must be taken to employ the type expression correctly. For example it would not be correct to write

```
rec "a" :: foo bar _rec%L
```

because `_rec` by itself is not a type expression but a function returning a type expression. Rather, it would be necessary to write

```
rec "a" :: foo bar (_rec "a")%L
```

or something similar.

It is not strictly necessary for the formal parameter of the type identifier to be the same as that of the whole declaration (although certain optimizations apply if it is). For example, a tree with node types alternating by levels could be declared as follows.

```
tree ("x","y") :: root "x" subtrees (_tree ("y","x"))%L
```

The argument to the type mnemonic `tree` and the type identifier `_tree` should always be a pair of type expressions.

Example

Listing 4.9 defines a first order parameterized record meant to model a polymorphic set type with an automatically initialized field maintaining the cardinality of the set. The parameter is a type expression giving the types of the elements. In one case a specialized form of the record is defined, with the element type fixed as real. In another case, the record with an element type of strings is invoked.

Assuming Listing 4.9 resides in a file `prec.fun`, we can exercise it as follows.

```
$ fun prec.fun --m=x --c realset_type
polyset(1%o&)[
   elements: {2.000000e+00,3.000000e+00,1.000000e+00},
   cardinality: 3]
$ fun prec.fun --m=y --c "_polyset %s"
polyset(1%oi&)[elements: {'bar','foo'},cardinality: 2]
```

The `1%oi&` parameter to the `polyset` record mnemonic is displayed as a reminder that the latter is a first order parameterized record. It can be seen that in each case, the set elements are displayed as instances of the corresponding parameter type.

4.3 Type stack operators

Some types and type induced functions remain problematic to specify in terms of the type expression features introduced hitherto. These include enumerated types, recursive types other than records or trees, tagged unions, and functions to generate random instances of a type. Where records are concerned, there is still a need to be able to combine two different record types given by symbolic names within a single binary constructor (e.g., a pair of records). These remaining issues are all addressed by a combination of some new type operators, and a new way of looking at type expressions documented in this section.

4.3.1 The type expression stack

To use type expressions to their fullest extent, it is necessary to understand them in more operational terms than previously considered. Previous examples have employed type expressions of the form `%`uvW, for a binary type constructor W and arbitrary type expressions u and v, referring to u as the left subexpression and v as the right. Equivalently, one could envision an automaton scanning forward through the expression and accumulating parts of it onto a stack. When W is reached, the left operand u will be at the bottom of the stack, and the more recently scanned right operand v will be at the top. W is then combined with the uppermost operands on the stack, coincidentally also its left and right subexpressions.

If type expressions really were scanned by an automaton that used a stack, then perhaps more flexible ways of building them would be possible. The initial contents of the stack could be chosen to order, and some direct control of the automaton could be requested when the expression is scanned. There is in fact a way of doing both of these.

Initializing the stack

It is mentioned on page 155 that a symbolic type expression (for example, a record type `_foobar`) can be combined with literal type operators (for example, the instance recognizer operator `I`) in a type expression such as `_foobar%I`. The symbolic name on the left of the `%` and the literals on the right are previously justified by syntactic necessity, but it is generally true that any expression x can be placed immediately to the left of a type

mnemonic	interpretation
`d`	duplicate the operand on the top of the stack
`l`	replace the top operand on the stack with its left side
`r`	replace the top operand on the stack with its right side
`w`	swap the top two operands on the stack

Table 4.2: type stack manipulation operators

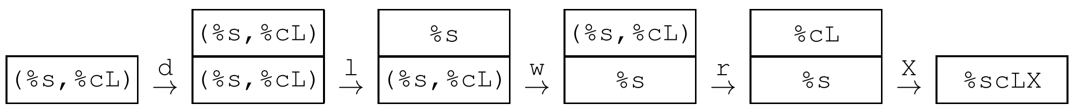


Figure 4.1: illustration of type stack evolution to evaluate `(%s,%cL)%dlwrX`

expression. In operational terms, the effect will be that x is pushed onto the otherwise empty stack before scanning begins.

Controlling the scanning automaton

With stack initialization settled, the issue of instructing the automaton is addressed by the four operators in Table 4.2. These operators can be seen as instructions addressed directly to the automaton like keystrokes on a calculator, rather than components of the type being constructed. There are some additional notes to the brief descriptions in the table.

- If the top value on the stack is a list rather than a pair, the `l` operator will extract its head and the `r` operator will extract its tail.
- If the top value is a triple rather than a pair, the `l` operator will extract the left side, and the `r` operator will extract the other pair of components. The latter can be further deconstructed by `l` or `r`.
- The above generalizes to n-tuples of the form $(x_0, x_1 \dots x_n)$, assuming no inner parentheses. On the other hand, a triple $((x, y), z)$ is treated as a pair whose left side is a pair.

Example

A simple example conveniently demonstrates all four type stack manipulations. The initial contents of the type stack will be the pair of type expressions `(%s,%cL)`, for strings and lists of characters respectively. Our task will be to write a type expression that manually constructs the product type `%scLX` from this configuration. Although this technique is unduly verbose for a pair of literal type expressions, it could also be used on a pair of symbolic type expressions, such as record type identifiers, for which there would be no alternative.

mnemonic	interpretation
B	record type constructor the hard way
Q	compressor function or compressed type constructor
i	random instance generator
h	recursive type or recursion order lifter
u	unit type constructor

Table 4.3: type operators with idiosyncratic usage

This task is easily accomplished by the sequence of operations `d`, `l`, `w`, and `r` in that order. An animation of the algorithm is shown in Figure 4.1. To confirm that this understanding is correct, we execute the following test.

```
$ fun --m="('foo','bar')" --c "(%s,%cL)%dlwrX"
('foo',<`b,`a,`r>)
$ fun --m="('foo','bar')" --c %scLX
('foo',<`b,`a,`r>)
```

With identical results in both cases, the types appear to be equivalent. To be extra sure, we can even do this,

```
$ fun --m="~&E(%scLX,(%s,%cL)%dlwrX)" --c %b
true
```

recalling that the `~&E` pseudo-pointer is for comparison.

Another variation shows that the subexpressions need not be used in the order they're written down, because the automaton can be instructed to the contrary.

```
$ fun --m="('foo','bar')" --c "(%s,%cL)%drwlX"
(<`f,`o,`o>,'bar')
```

However the original way is less confusing.

The pattern `dlwr` is needed so frequently in type expressions that it is inferred automatically when the literal portion of a type expression begins with a binary constructor.

```
$ fun --m="~&E((%s,%cL)%X,(%s,%cL)%dlwrX)" --c %b
true
```

Remembering this convention can save a few keystrokes.

4.3.2 Idiosyncratic type operators

A small selection of type operators remaining to be discussed is documented in this section, which is shown in Table 4.3. All of these rely in some essential way on an appropriately initialized type stack in order to be useful, and therefore depend on the preceding discussion as a prerequisite.

`B` – Record type constructor

A type expression of the form x`%B` represents a record type. If it is used explicitly instead of declaring a record the normal way, then x should be a list of the form

$$
\begin{array}{lll}
< & & \\
& \langle \textit{record mnemonic} \rangle \textbf{:} & \langle \textit{initializer} \rangle, \\
& \langle \textit{field identifier} \rangle \textbf{:} & \langle \textit{type expression} \rangle, \\
& \vdots & \vdots \\
& \langle \textit{field identifier} \rangle \textbf{:} & \langle \textit{type expression} \rangle >
\end{array}
$$

where the record mnemonic and field identifiers are character strings, and the initializer is a function to initialize the record. This function must be consistent with the conventions for record initializing functions explained in Section 4.2.3 and with the types and initializing functions of the subexpressions, as well as their number and memory map.

This type constructor never has to be used explicitly because the compiler does a good job of generating record type expressions automatically from record declarations. It exists as a feature of the language only to establish a semantics for record declarations in terms of a quasi-source level transformation. Users are advised to let the compiler handle it.

`Q` – Compressor function or compressed type constructor

There are several ways of using the `Q` type operator as previously noted on pages 131 and 140. One way is in specifying the type expressions of compressed types, another is in specifying a function that uncompresses an instance of a compressed type, and another is as a compression function. Examples are `%sLQ` for the type of compressed lists of character strings, `%sLQI` for the instance recognizer and extraction function of compressed lists of character strings, and `%Q` for the (untyped) compression function.

In view of type expressions as stacks, it would be equivalent to write t`%Q` or t`%QI` respectively for the compressed form or extraction function of a type t. There is also a more general form of compression function, n`%Q`, where n is a natural number. Note that this usage is disambiguated from t`%Q` by n being a natural number and t being a type expression.

Granularity of compression The number n specifies the granularity of compression. Higher granularities generally provide less effective but faster compression. The compression algorithm works by factoring out common subtrees in its argument where doing so can result in a net decrease in space. The granularity n is the size measured in quits of the smallest subtree that will be considered for factoring out.

Choice of granularity Anything with significant redundancy can be compressed with a granularity of 0, equivalent to `%Q` with no parameter. If faster compression is preferred, the best choice of granularity is data dependent. Granularities on the order of 10^3 quits or more are conducive to noticeably faster compression, but not always applicable. For example, to compress a function of the form $h(f, f)$ where f is a large function or constant appearing twice in the function be compressed, a granularity larger than the size of f would be ineffective. A granularity equal to the size of f or slightly smaller would cause f to be factored out and nothing else, assuming it is the largest repeated subexpression. (The size of f can be determined by displaying it in opaque format or by the `weight` function.)

`i` – Random instance generator

The `i` type operator generates a function that generates random instances of a given type. Some comments relevant to the `i` operator are found on page 130 in relation to the semantics of the printed format of opaque types, because they are printed as an expression that includes the `i` operator, but the present aim is to document the `i` operator specifically and in detail.

Usage In terms of the stack description of type expressions, the `i` operator requires two operands on the stack, with the top one being a type expression and the one below being a natural number. A simple way of using it is therefore by an expression of the form `(`n`,`t`)%i` for a natural number n and a symbolic type expression t, or more concisely n`%`u`i` if the type can be expressed as a sequence of literals u. The former relies on the convention of an implicit `dlwr` inserted before the `i` as mentioned on page 168.

Size of generated data The natural number n usually represents the size measured in quits of the random data that the function will generate. In some cases the size is inapplicable or only approximate because the concrete representation of the type instances constrains it. For example, boolean values come in only two sizes. However, a size must always be specified.

In one other case, namely expresions of the form n`%cOi` with n less than 256, the number n represents the ISO code of the character that is generated if the function is applied to the argument `&`. That is, the function behaves deterministically when applied to `&` but returns a random character otherwise.

Semantics of generating functions Other than as noted above, random instance generators ignore their arguments, hence the usual idiomatic practice of writing n`%`u`i&` to express a random compile-time constant, wherein the argument is `&`. An alternative would be for the argument to influence the statistical properties of the result, but to do so in any more than an *ad hoc* way is a matter for further research by compiler developers.

Consequently, there is no way of controlling the distribution of results obtained by random instance generators other than by post-processing (although the language provides other ways to generate random data that are more controllable). Some rough guidelines about the (hard coded) statistics used by instance generators are as follows.

- Floating point numbers of type `%e` or `%E` are uniformly distributed between -10 and 10.

- Complex numbers (type `%j`) have their real and imaginary parts uncorrelated and uniformly distributed between -10 and 10.
- Strings, natural numbers and most aggregate types such as lists and sets have their length chosen by a random draw from a uniform distribution whose upper bound increases logarithmically with n. The sizes of the elements or items are then chosen randomly to make up the total required size.
- Raw data, transparent types, trees, and functions are generated by an *ad hoc* algorithm to achieve a qualitative mix of tree shapes.

Properly speaking, random instance generators are not functions at all, and do not sit comfortably within the functional programming paradigm. Some comments on the `~&K8` pseudo-pointer in Section 2.5.1 are applicable here as well.

Example To generate an arbitrary module of dual type trees of characters and natural numbers for stress testing a function that operates on such types, the following expression can be used.

```
$ fun --m="500%cnDmi&" --c %cnDm
<
   'QMS': `U^: <
      0^: <>,
      `P^: <8^: <>,14^: <>,0^: <>,6^: <>>,
      ^: (
         149%cOi&,
         <2^: <>,~&V(),1^: <>,0^: <>,0^: <>>),
      2^: <>>,
   '{V}gamO$`': 244%cOi&^: <218%cOi&^: <24^: <>>,2^: <>>,
   '?xtyv9kN#/AJ': 2^: <>,
   'P9tPxo[_': 220%cOi&^: <~&V(),0^: <>,4^: <>>,
   '-/.X-D+g`Y': `P^: <0^: <>>>
```

See page 130 for more examples.

Limitations Due to issues with non-termination, random instance generators apply only to non-recursive types (i.e., those that don't involve the `h` operator or circular record declarations). A diagnostic message of "`bad i type`" is reported if it is used with a recursive type.

`h` – Recursive type or recursion order lifter

The recursive type operator `h` can be used to specify the types of self-similar data structures. Normally tree types (`%`x`T` and `%`x`D`) or recursively defined records (page 158) are sufficient for this purpose, but this type constructor facilitates unrestricted patterns of self-similarity if preferred, and with less source level verbiage than a record.

Semantics This operator can be understood only in terms of the type expression stack, because its arity is variable. If the top of the stack already contains an `h`, then the next `h` is combined with it like a unary operator, but otherwise it serves as a primitive. The `h` operator is not meaningful in itself, but its presence in a type expression implies the validity of certain semantics preserving rewrite rules by definition.

- If an `h` appears without any `h` adjacent to it, the innermost subexpression containing it may be substituted for it.
- If a consecutive sequence of n of them appears without another `h` adjacent to it, the sequence can be replaced by the subexpression terminated by the n-th type operator following the sequence, numbering from 1. This rule is a generalization of the previous one.

These rewrite rules always lengthen a type expression and never lead to a normal form, but the intuition is that they allow a type expression to be expanded as far as needed to match a given data structure.

Examples The simplest example of a recursive type is `%hL`. This is the type of lists of nothing but more lists of the same. It is equivalent to `%hLL`, and to `%hLLL`, and so on. Anything can be cast to this type.

```
$ fun --m="0" --c %hL
<>
$ fun --m="&" --c %hL
<<>>
$ fun --m="'foo'" --c %hL
<
   <<<>>,<<>,<>>>,
   <<<>>,<<>,<<>,<>>>>,
   <<<>>,<<>,<<>,<>>>>>
```

The next simplest example is the type of nested pairs of empty pairs, `%hhWZ`. Because there are two consecutive recursive type constructors, this type is equivalent to `%hhWZWZ`, and so on.

```
$ fun --m="0" --c %hhWZ
()
$ fun --m="(&,&,0)" --c %hhWZ
(((),()),((),()),())
```

For a more complicated example, a type of binary trees of strings is constructed using assignment of strings to pairs of the type. The trees are expressed in the form

$$\langle root \rangle\texttt{:}\ \ \texttt{(}\langle left\ subtree \rangle\texttt{,}\langle right\ subtree \rangle\texttt{)}$$

The empty tree is `()`, a tree with only one node is `'a': ()`, a tree with two empty subtrees is `'b': ((),())`, and so on. The type expression is `%shhhhWZAZ`.

```
$ fun --m="'a': ('b': ('c': (),'d': ()),())" --c %shhhhWZAZ
'a': ('b': ('c': (),'d': ()),())
```

u – Unit type constructor

These types have only a single instance, and are expressed by a type expression of the form ⟨*instance*⟩`%u`. For example, the type containing only the true boolean value could be expressed `true%u`.

The printing function for a unit type prints the instance in general (`%g`) form. Because printing functions don't check the validity of their arguments, they will print the instance even if the argument is something other than that. However, the `--cast` command line argument will detect a badly typed argument.

Unit types have a default value when declared as the type of a field in a record. The default value is the instance. The field will be automatically initialized to the instance when the record is created.

Tagged unions A good use for unit types is to express tagged unions, which could be done by an expression such as `(0%unX,&%usX)%U` for a tagged union of naturals (`%n`) and strings (`%s`), using boolean values (`0` and `&`) as the tags. Naturals, characters, and strings also make good tags. The tag field could be on the left or the right side of a pair, but more efficient code is generated when the tag field is on the left, as shown above.

A tagged union avoids the possibility of ambiguity characteristic of free unions by ensuring that the instances of the subtypes of the union have disjoint sets of concrete representations. For example, the empty tree `()` could represent either the natural number `0` or the empty string, `''`, but the tag value determines the intended interpretation.

```
$ fun --main="(0,())" --c "(0%unX,&%usX)%U"
(0,0)
$ fun --main="(&,())" --c "(0%unX,&%usX)%U"
(&,'')
```

Enumerated types Another use for unit types is to construct enumerated types by forming the free union of a collection of them. The benefits of an enumerated type are that the instance checker can automatically verify membership, so records with enumerated types for their fields have built in sanity checking and initialization. The default value of a field declared as an enumerated type is an arbitrary but fixed instance, depending on the order they are given in the type expression.

An example of an enumerated type for weekdays would be

```
((((('mon'%u,'tue'%u)%U,'wed'%u)%U,'thu'%u)%U,'fri'%u)%U
```

A more elegant and more efficient way of expressing it would be

```
enum block3 'montuewedthufri'
```

using functions introduced subsequently. The instance checker can be seen to work as expected.

```
$ fun --m="(enum block3 'montuewedthufri')%I 'mon'" --c %b
true
$ fun --m="(enum block3 'montuewedthufri')%I 'sun'" --c %b
false
```

On the other hand, if the concrete representation of an enumerated type is of no consequence but symbolic names for the instances would be convenient, then a simpler way to declare one would be to use the field identifiers from a record declaration instead of character strings, as in `weekdays :: mon tue wed thu fri`. A further declaration along these lines

```
weekday_type = enum <mon,tue,wed,thu,fri>
```

would allow `weekday_type` to be used as an ordinary type expression, but the displayed format of a value cast to this type would be more difficult to interpret than one with strings as a concrete representation.

4.4 Remarks

This chapter in combination with the previous one brings to a close all necessary preparation to use type expressions and related features effectively in Ursala. You are welcome to take it cafeteria style, because in this language types are your servant rather than your master (barring BWI alerts to the contrary).

Although type expressions are first class objects in the language, we have avoided discussion of their concrete representations, because they are designed to be treated as opaque. As one author aptly put it, “the type of type is type”. Readers wishing to know more about how they are implemented are referred to Part IV of this manual on compiler internals.

If any of this material is difficult to remember, a quick reminder can be obtained by the command `$ fun --help types` whose output is shown in Listing 4.10.

Listing 4.10 output from `$ fun --help types`

```
type stack operators of arity 0
-------------------------------
E  push primitive arbitrary precision floating point type
a  push primitive address type
b  push primitive boolean type
c  push primitive character type
e  push primitive floating point type
f  push primitive function type
g  push primitive general data type
j  push primitive complex floating point type
n  push primitive natural number type
o  push primitive opaque type
q  push primitive rational type
s  push primitive character string type
t  push primitive transparent type
x  push primitive raw data type
y  push primitive self-describing type

type stack operators of arity 1
-------------------------------
B  construct a record type from a module
C  transform top type to exceptional input printing wrapper
G  transform top type to recombining grid thereof
I  transform top type to instance recognizer
J  transform top type to job thereof
L  transform top type to list thereof
M  transform top type to error messenger
N  transform top type to balanced tree thereof
O  make top type printed as opaque
P  transform top type to printing function
Q  transform top type to compressed version
R  qualify C or V with recursive attribute
S  transform top type to set thereof
T  transform top type to a tree thereof
W  transform top type to a pair
Y  transform top type to self-describing formatter
Z  replace top type with union with empty instance
d  duplicate the operand on the top of the stack
h  push recursive type or raise the top one
k  transform top type or function to identity function
l  replace the top operand on the stack with its left side
m  transform top type to list of assignments of strings thereto
p  transform top type to parsing function
r  replace the top operand on the stack with its right side
u  transform top constant to unit type

type stack operators of arity 2
-------------------------------
A  transform top two types type to an assignment
D  replace top two types with dual type tree
U  replace top two types with free union thereof
V  transform top types to i/o validation wrapper generator
X  transform top two types type to a pair
i  transform top type to random instance generator
w  swap the top two operands on the stack
```

Just say to me "you're going to have to do a whole lot better than that", and I will.

Harrison Ford in *Mosquito Coast*

5 Introduction to operators

Most programs in Ursala attain their prescribed function through an algebra of functional combining forms. Its terms derive from the dozens of library functions and endless supply of user defined primitives documented elsewhere in this manual, along with a versatile repertoire of operators addressed in this chapter and the succeeding one. As the key to all aspects of flow and control, a ready command of these operators is no less than the essence of proficiency in the language.

Although all features of the language are extensible by various means, in normal usage the operators are regarded as a fixed set, albeit a large one. There are about a hundred operators, most of which are usable in prefix, infix, postfix, and nullary forms, and many of them further enhanced by optional suffixes modifying their semantics.

Because operators are a broad topic, they are covered in two chapters. This chapter discusses conventions pertaining to operators in general, followed by detailed documentation of the more straightforward class of so called aggregate operators. The next chapter catalogs the full assortment of the remaining available operators in groups related by common themes as far as possible.

The design of the language favors a pragmatic choice of operators over aesthetic notions of orthogonality. Any operator described here has earned its place by being useful in practice with sufficient frequency to warrant the mental effort of remembering it.

5.1 Operator conventions

This section briefly documents some general conventions regarding operator syntax, arity, precedence, and algebraic properties.

<table>
<tr><th>suffix</th><th>applicable stems</th></tr>
<tr><td>pointers</td><td>& := -> ^= $ ~* * |\ ^ ^~ ^| ^* ? ^? ?= ?< *~ != -< *| ~| |=</td></tr>
<tr><td>opcodes</td><td>.. .| .!</td></tr>
<tr><td>types</td><td>% %-</td></tr>
<tr><td>|</td><td>/ \</td></tr>
<tr><td>~</td><td>^~ ^| ^*</td></tr>
<tr><td>$</td><td>/ \ /* * + ;</td></tr>
<tr><td>*</td><td>/ \ /* * + ; *= ^~ ^| ^* *^ %= |=</td></tr>
<tr><td>-</td><td>%=</td></tr>
<tr><td>.</td><td>+ ; *^</td></tr>
<tr><td>;</td><td>/ \</td></tr>
<tr><td><</td><td>^?</td></tr>
<tr><td>=</td><td>/* * + ; *= ^~ ^| ^* ^? *^ %= |=</td></tr>
</table>

Table 5.1: suffixes and their operator stems

5.1.1 Syntax

Syntactically an operator consists of a stem followed by a suffix. The stem is expressed by non-alphanumeric characters or punctuation marks. These characters are not valid in user defined function names or other identifiers. The most frequently used operators have a stem of a single character, such as + or `:`. However, there aren't enough non-alphanumeric characters to allow a separate one for each operator, so some operator stems are expressed by two consecutive characters, such as `^:` and `|=`. These character combinations when used as an operator stem are treated in every way as indivisible units, just as if they were a single character.

The suffix of an operator may contain alphanumeric or non-alphanumeric characters, depending on the operator. Lexically the stem and the suffix are nevertheless an indivisible unit.

Use of suffixes

The suffix modifies the semantics of an operator, usually in some small way. For example, an expression like `f+g` represents the composition of functions `f` and `g`, but `f+*g`, with a suffix of `*` on the composition operator, is equivalent to `map f+g`, the function that applies `f+g` to every item of a list.

Not all operators allow suffixes, and among those that do, the effect of the suffixes varies. Two illustrative examples familiar from previous chapters involving operators with suffixes are `&` and `%`, for pseudo-pointers and type expressions. Quite a few operators allow pointer expressions as suffixes, as shown in Table 5.1, and they use them in different ways.

Further lexical conventions

Because operator characters are not valid in identifiers, operators and identifiers can be adjacent without intervening white space and without ambiguity. In fact, omitting white space is often a requirement for reasons to be explained presently.

A possibility of ambiguity arises when operators are written consecutively, or when an operator with an alphanumeric suffix is followed immediately by an identifier. Lexically the ambiguity is always resolved in favor of the left operator at the expense of the right. For example, `/` and `*` are both operators, but so is `/*`, and this character combination is interpreted as the latter operator rather than a juxtaposition of the other two.

In rare cases where a juxtaposition without space is semantically necessary but syntactically ambiguous, the expressions can be parenthesized.

5.1.2 Arity

There are four possible arities for most operators, which are prefix, postfix, infix, and solo (nullary). An infix operator takes two operands and is written between them. Prefix and postfix operators take one operand and are written before or after it, respectively. A solo operator takes no operands as such, but may be used as a function or as the operand of another operator. Aggregate operators such as parentheses and brackets are outside this classification, and some operators do not admit all four arities.

Disambiguation

It is important to be precise about the arity intended for any usage of an operator, because the semantics may differ between different arities of the same operator, and no general rule relates them. For operators admitting only one arity, there is no ambiguity, but otherwise the usual way of distinguishing between arities of an operator is by its proximity to any operands in the source text.

- If an operator can be either infix or something else, then the infix arity is implied precisely when the operator is immediately preceded and followed by operands with no intervening white space or comments, as in `f+g`.
- If infix usage is ruled out but the operator admits a postfix form, the postfix usage is implied whenever the operator is immediately preceded by an operand, as in `f*`.
- If both the infix and postfix usages can be excluded but prefix and solo usages are possible, the determination in favor of the prefix usage is indicated by an operand immediately following the operator, as in `~p`.

The crucial observation should be that white space affects the interpretation. An expression like `f=>y` has a different meaning from `f=> y`, because the `=>` is interpreted as infix in the first case and postfix in the second. These conventions differ from other modern languages, wherein white space plays no rôle in disambiguation.

Pathological cases

Although the rules above are not completely rigorous, a real user (as opposed to a compiler developer) should view arity disambiguation this way most of the time, and parenthesize an expression fully when in doubt. Doubts might occur in the case of an operator in its solo usage being the operand of another operator. For example, the ~ and + operators both allow solo usage, the ~ can also be prefix, and the + can also be postfix, so does ~+ mean `(~)+` or `~(+)`? It's best to settle the issue by writing one of the latter.

On the other hand, some may consider parentheses an unsightly and unwelcome intrusion, and some may insist on a clear convention as a matter of principle. The latter are referred to Part IV of this manual, while the former may find it convenient to ask the compiler whether it will parse the expression the way they intend.

```
$ fun --m="~+" --parse
main = (~)+
```

The output from the `--parse` option shows the main expression fully parenthesized, and is useful where operators are concerned. The alternative parsing, incidentally, would not be sensible for these particular operators, and on that score the compiler usually gets it right.

5.1.3 Precedence

Operator precedence rules settle questions of whether an expression like `x+y/z` is parsed as `x+(y/z)` or `(x+y)/z`. The parsing that is most intuitive to a person who has learned to think in Ursala turns out to require fairly complicated rules when formally codified. An operator precedence relation exists, but it is neither transitive, reflexive, nor antisymmetric. For a given pair of operators, the relationhip may also depend on the way their arities are disambiguated.

The intuitive approach

The easiest way to cope with operator precedence when learning the language is to write most expressions fully parenthesized at first, and wait for habits to develop. For example, instead of writing `f+g*` for the composition of `f` with the map of `g`, write `f+(g*)` so there is no mistaking it for `(f+g)*`. In time, it may become noticeable that the usage `f+(g*)` occurs more frequently in practice than `(f+g)*`. It then becomes meaningful to ask whether the compiler does the "right thing", by parsing it the way it would usually be intended.

```
$ fun --m="f+g*" --parse
main = f+(g*)
```

There's a good chance that it does, because the precedence rules were developed from observations of usage patterns. In cases where it accords with intuition, one may choose to drop the habit of fully parenthesizing expressions of that form, until eventually parentheses are used only when necessary.

In combination with this learning approach, two operator precedence rules are important enough to be committed to memory from the outset, or it will be difficult to make any progress.

`*|`	`$`	`!=`	`%`	`-`	`*^`	`?`	`+`	`!|`	`/`	`*-`	`:`	`::`
`-<`	`$-`	`%=`	`%~`	`..`	`:-`	`?$`	`;`	`&&`	`/*`	`-*`	`^:`	`=`
`->`	`$^`	`*~`	`^`		`<:`	`?<`	`^=`	`-=`	`\`	`--`		
`|=`	`*`	`-~`	`^*`		`=>`	`?=`		`==`	`\*`	`|`		
`~|`	`*=`	`=:`	`^|`			`^?`		`^!`				
	`-$`	`=]`	`^~`			`|\`		`^&`				
	`-:`	`[=`						`||`				
	`.!`	`~-`						`~<`				
	`.|`							`~=`				
	`@`											
	`~*`											
	`~~`											

Table 5.2: each operator in the table is equivalent in precedence to its column header

- Function application, when expressed by juxtaposition with white space between the operands, has lower precedence than almost everything else and is right associative. Hence `f+g u/v x` parses as `(f+g) ((u/v) x)`.
- Function application expressed by juxtaposition without intervening white space has higher precedence than almost everything else and is left associative. Hence the expression `g+f(n)x` is parsed as `g+((f(n))x)`.

The operators having lower precedence than application in first case are only things like commas, parentheses, and declaration operators. The only exception to the second rule is the prefix tilde `~` operator. Associativity is not a separate issue from precedence, because it's a consequence of whether an operator has lower precedence than itself.

Experienced functional programmers might observe that right associativity of function application will seem unconventional to them, but they are outnumbered by mathematicians, engineers, and scientists other than quantum physicists. Those who take issue are asked to consider whether the alternative of left associativity would make much sense in a language without automatic currying.

The formal approach

For the benefit of compiler developers, bug hunters, and language lawyers, and to prove that such a thing exists, a complete account of precedence rules for all infix, prefix, and postfix operators other than function application is given by Tables 5.2 through 5.6.

Equivalent precedences Operators are partitioned into seventeen equivalence classes with respect to precedence. The classes with multiple members are shown in Table 5.2. The remaining tables are expressed in terms of a representative member from each class.

There are four operator precedence relations, each applicable to a different context, and each depicted in a separate one of Tables 5.3 through 5.6. Precedence relationships for

	$\star\vert$	`$`	`!=`	`%`	`−`	`*^`	`?`	`+`	$!\vert$	`:=`	`~`	`.`	`/`	`!`	`*−`	`:`	`::`
$\star\vert$		●	●		●	●		●	●	●		●			●	●	
`$`					●							●			●	●	
`!=`		●		●	●					●		●			●	●	
`%`		●			●					●		●			●	●	
`−`																	
`*^`		●	●		●				●	●		●			●	●	
`?`																	
`+`		●	●	●	●	●			●	●		●	●		●	●	
$!\vert$		●	●	●	●					●		●			●	●	
`:=`		●			●							●			●	●	
`~`																	
`.`					●												
`/`	●	●	●	●	●	●		●	●	●		●			●	●	
`!`																	
`*−`					●							●				●	
`:`					●							●					
`::`	●	●	●	●	●	●		●	●	●		●	●		●	●	

Table 5.3: infix-infix operator precedence relation

	$\star\vert$	`$`	`!=`	`%`	`−`	`*^`	`?`	`+`	$!\vert$	`:=`	`~`	`.`	`/`	`!`	`*−`	`:`	`::`
$\star\vert$			●		●	●			●	●		●		●	●	●	
`$`					●							●		●	●	●	
`!=`				●	●					●		●		●	●	●	
`%`																	
`−`																	
`*^`			●		●				●	●		●		●	●	●	
`?`																	
`+`																	
$!\vert$		●		●	●					●		●		●	●		
`:=`		●			●							●		●	●	●	
`~`					●							●					
`.`																	
`/`																	
`!`																	
`*−`					●							●				●	
`:`					●							●					
`::`																	

Table 5.4: prefix-postfix operator precedence relation

	⋆|	$	!=	%	−	⋆^	?	+	!|	:=	~	.	/	!	⋆−	:	::
⋆|		●	●		●	●		●	●	●		●			●	●	
$					●							●			●	●	
!=		●		●	●					●		●			●	●	
%																	
−																	
⋆^		●	●		●				●	●		●			●	●	
?																	
+																	
!|		●	●	●	●					●		●			●	●	
:=		●			●							●			●	●	
~					●							●				●	
.																	
/																	
!																	
⋆−					●							●				●	
:					●							●					
::																	

Table 5.5: prefix-infix operator precedence relation

	⋆|	$	!=	%	−	⋆^	?	+	!|	:=	~	.	/	!	⋆−	:	::
⋆|			●		●	●			●	●		●		●	●	●	
$			●		●	●			●	●		●		●	●	●	
!=				●	●					●		●		●	●	●	
%					●					●		●		●	●	●	
−																	
⋆^			●		●				●	●		●		●	●	●	
?																	
+		●	●	●	●	●				●		●		●	●		
!|				●	●					●		●		●	●		
:=					●							●		●	●	●	
~																	
.					●												
/		●	●	●	●	●				●		●		●	●		
!																	
⋆−					●							●				●	
:					●							●					
::	●	●	●	●	●	●	●	●	●	●		●		●	●	●	

Table 5.6: infix-postfix operator precedence relation

operators not shown in Tables 5.3 through 5.6 can be inferred by their equivalence to those that are shown based on Table 5.2.

How to read the tables Each occurrence of a bullet in a table indicates for the relevant context that the operator next to it in the left column has a "lower" precedence than the operator above it in the top row. However, precedence is not a total order relation. Two operators can be unrelated, or can be "lower" than each other. To avoid confusion, it is best simply to refer to one operator as being related to another by the precedence relation, and to assume nothing about a relationship in the other direction.

- Table 5.3 pertains to precedence relationships between infix operators. If an infix operator $\oplus$ from the left column is unrelated to an infix operator $\otimes$ from the top row (i.e., if a bullet is absent from the corresponding position), then an expression $x \oplus y \otimes z$ will be parsed as $(x \oplus y) \otimes z$. Otherwise, it will be parsed as $x \oplus (y \otimes z)$.
- Table 5.4 pertains to precedence relationships between prefix and postfix operators. If a prefix operator $\triangle$ from the left column is unrelated to a postfix operator $\triangledown$ from the top row, then an expression $\triangle\, x\triangledown$ will be parsed as $(\triangle\, x)\triangledown$ Otherwise, it will be parsed as $\triangle\,(x\triangledown)$.
- Table 5.5 pertains to relationships between prefix and infix operators. If a prefix operator $\triangle$ from the left column is unrelated to an infix operator $\oplus$ from the top row, then an expression $\triangle\, x \oplus y$ will be parsed as $(\triangle\, x) \oplus y$. Otherwise, it will be parsed as $\triangle\,(x \oplus y)$.
- Table 5.6 pertains to relationships between infix and postfix operators. If an infix operator $\oplus$ from the left column is unrelated to a postfix operator $\triangledown$ from the top row, then an expression $x \oplus y\triangledown$ will be parsed as $(x \oplus y)\triangledown$. Otherwise, it will be parsed as $x \oplus (y\triangledown)$.

5.1.4 Dyadicism

Although a given operator may have different meanings depending on the way its arity is disambiguated, in many cases the meanings are related by a formal algebraic property. The word "dyadic" is used in this manual to describe operators that allow an infix arity and have certain additional characteristics.

- If an operator $\circ$ has a solo and an infix arity, and it meets the additional condition $(\circ)\ (a, b) = a \circ b$ for all valid operands a and b, then it is called solo dyadic.
- If an operator $\circ$ allows a prefix and an infix arity such that $(\circ b)\ a = a \circ b$, then it is called prefix dyadic.
- If an operator $\circ$ admits a postfix and an infix arity, and satisfies $(a\circ)\ b = a \circ b$, then it is called postfix dyadic.

Motivation for dyadic operators

Determining the dyadicism of a given operator in this sense obviously is not computable, so the property or lack thereof is recorded for each operator by a table internal to the compiler. This information permits certain code optimizations, and also reduces the bulk of reference documentation. Where an operator is noted to be dyadic, the semantics for the dyadic arity may be inferred from that of the infix, and need not be explicitly stated.

Dyadic operators also make the language easier to use. If an expression like `f+g:-k` is required, and the intended parsing is `f+(g:-k)`, another alternative to parenthesizing it, remembering the precedence rules, or checking them with the `--parse` option is to remember that the composition operator (+) is postfix dyadic. The expression therefore can be rewritten as `f+ g:-k` consistently with its intended meaning. The space represents function application, which has the lowest precedence of all, so the expression can only be parsed as `(f+) (g:-k)`.

If the intended parsing is `(f+g):-k`, which would not be the default under the precedence rules, there is still an alternative. Using the fact that the reduction operator (`:-`) is prefix dyadic, we can rewrite the expression as `:-k f+g`.

Table of dyadic operators

Most operators are dyadic in one form or another, especially postfix, so it may be easier to remember the counterexamples, such as the folding operator, =>. The following table lists the arities and dyadicisms for all infix, prefix, postfix, and solo operators in the language other than function application and declaration operators.

Table 5.7: Operator arities and algebraic properties

	arity				dyadicism		
mnemonic	prefix	infix	postfix	solo	prefix	postfix	solo
`:`	●	●	●	●	●	●	●
`^:`	●	●	●	●	●	●	●
`|`		●		●			●
`--`	●	●	●	●	●	●	●
`-*`	●	●	●	●	●	●	●
`*-`	●	●	●	●			
`!`			●	●			
`/`		●		●			●
`\`		●		●			●
`/*`		●		●			●
`\*`		●		●			●
`&`				●			
`@`			●	●			
`.`		●	●	●			
`~`	●			●			
`:=`	●	●	●	●	●	●	●
`&&`	●	●	●	●	●	●	●

Table 5.7: Operator arities and algebraic properties (continued)

<table>
<tr><th></th><th colspan="4">arity</th><th colspan="3">dyadicism</th></tr>
<tr><th>mnemonic</th><th>prefix</th><th>infix</th><th>postfix</th><th>solo</th><th>prefix</th><th>postfix</th><th>solo</th></tr>
<tr><td>||</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td></tr>
<tr><td>!|</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td></tr>
<tr><td>^&</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td></tr>
<tr><td>^!</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td></tr>
<tr><td>-=</td><td>•</td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td></td></tr>
<tr><td>==</td><td>•</td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td></td></tr>
<tr><td>~<</td><td>•</td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td></td></tr>
<tr><td>~=</td><td>•</td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td></td></tr>
<tr><td>-></td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td></tr>
<tr><td>^=</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>+</td><td></td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td>•</td></tr>
<tr><td>;</td><td></td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td></td></tr>
<tr><td>|\</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>~~</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>$</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>~*</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>*</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>*=</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>^</td><td></td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td></td></tr>
<tr><td>^~</td><td></td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td></td></tr>
<tr><td>^|</td><td></td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td></td></tr>
<tr><td>^*</td><td></td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td></td></tr>
<tr><td>?</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>^?</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>?=</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>?$</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>?<</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>=></td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td></td><td>•</td></tr>
<tr><td>:-</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td></tr>
<tr><td><:</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td></tr>
<tr><td>*^</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td></td><td></td></tr>
<tr><td>-</td><td></td><td>•</td><td></td><td></td><td></td><td></td><td></td></tr>
<tr><td>..</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>.|</td><td></td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td></td></tr>
<tr><td>.!</td><td></td><td>•</td><td>•</td><td>•</td><td></td><td>•</td><td></td></tr>
<tr><td>%</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>%~</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>%-</td><td></td><td></td><td></td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>-$</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td></tr>
<tr><td>-:</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td>•</td><td></td></tr>
<tr><td>=:</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>_~</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>~_</td><td></td><td></td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>*~</td><td></td><td>•</td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
<tr><td>!=</td><td>•</td><td>•</td><td>•</td><td>•</td><td></td><td></td><td></td></tr>
</table>

operators	meaning
`-?...?-`	cumulative conditional with default last
`-+...+-`	cumulative functional composition
`-|...|-`	cumulative short circuit functional disjunction
`-!...!-`	cumulative logical valued short circuit functional disjunction
`-&...&-`	cumulative short circuit functional conjunction
`[...]`	record or a-tree delimiters
`<...>`	list delimiters
`{...}`	set delimiters
`(...)`	tuple delimiters
`-[...]-`	text delimiters

Table 5.8: aggregate operators; each encloses a comma separated sequence of expressions

Table 5.7: Operator arities and algebraic properties (continued)

	arity				dyadicism		
mnemonic	prefix	infix	postfix	solo	prefix	postfix	solo
`%=`	•	•	•	•	•	•	•
`=]`	•	•	•	•		•	
`[=`	•	•	•	•		•	
`$^`	•	•	•	•			
`$-`	•	•	•	•			
`-<`	•	•	•	•			
`*|`	•	•	•	•			
`~|`	•	•	•	•			
`|=`	•	•	•	•			

5.1.5 Declaration operators

Two infix operators whose discussion is deferred are `::` and =.

- The `::` is used only for record declarations, and is explained thoroughly in the previous chapter.
- The = is used only for declarations other than records. It can appear at most once in any expression, and only at the root. It is better understood as a syntactically sugared compiler directive than an operator. Rather than computing a value, it effects a compile-time binding of a value to an identifier.

Declarations are discussed further in a subsequent chapter regarding their interactions with name spaces and output-generating compiler directives.

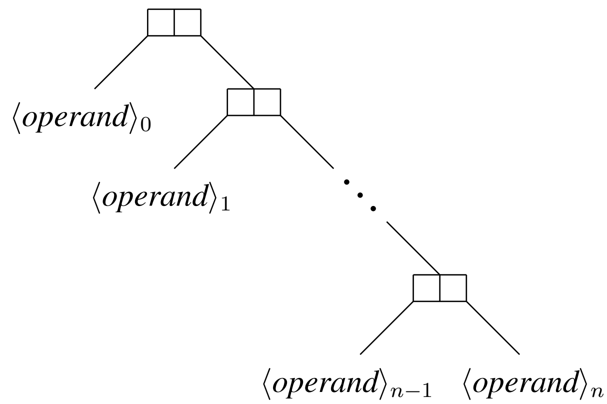


Figure 5.1: representation of a tuple `(`$\langle operand \rangle_0$`,`$\langle operand \rangle_1$`,` $\dots\langle operand \rangle_n$`)`

5.2 Aggregate operators

The operators listed in Table 5.8 are usable only in matching pairs, and with the exception of the text delimiters, `-[`...`]-`, they enclose a comma separated sequence of arbitrarily many expressions. With each enclosed expression serving as an operand, considerations of arity and precedence are not relevant to aggregate operators, but they employ a common convention regarding suffixes, as explained presently.

5.2.1 Data delimiters

The essential concepts of records, a-trees, lists, sets, tuples, and text follow from previous chapters, where the data delimiter operators in Table 5.8 are each introduced purely as a concrete syntax for one of these containers. When viewed as operators in their own right, they transform the machine representations of their operands to that of data structure containing them.

`()` – Tuple delimiters

On the virtual machine level, everything is represented either as an empty value or a pair. This representation directly supports the tuple delimiters, `(`...`)`. An empty tuple, `()`, maps to the empty value. If there is only one operand, the representation of the tuple is that of the operand. Otherwise, the representation is a pair with the first operand on the left and the representation of the tuple containing the remaining operands on the right, as shown in Figure 5.1.

`<>` – list delimiters

The list delimiters work similarly to the tuple delimiters except that a distinction is made between a singleton list and its contents. An empty list maps to the empty value, and any

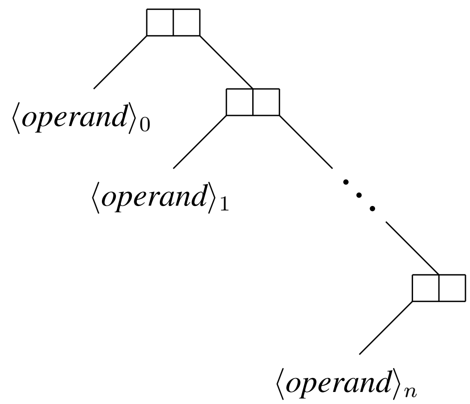


Figure 5.2: representation of a list <⟨*operand*⟩$_0$, ⟨*operand*⟩$_1$, ... ⟨*operand*⟩$_n$>

other list maps to the pair with the head on the left and the tail on the right. Equivalently, a list representation is like a tuple in which the last component is always empty, as shown in Figure 5.2.

{} – set delimiters

The set delimiters perform the same operation as the list delimiters, followed by the additional operation of sorting and removing duplicates. The sorting is done by the lexical order relation on characters and strings (regardless of the element type).

[] – record or a-tree delimiters

For these operators, each operand is expected to be an assignment of the form

⟨*address*⟩`:` ⟨*value*⟩

or equivalently a pair of an address and a value. The address is normally of the `%a` type, which is to say that its virtual machine representation has at most a single descendent at each level of the tree, as shown in Figure 5.3. (Branched addresses can be used if the associated data are a tuple of sufficient arity, as noted on page 154). The result is a structure in which each value is stored at a position that can be reached by following a path from the root described by the corresponding address.

Figure 5.3 provides a simple illustration of this operation. The structure created by the record delimiter operators from the given data contains the value ⟨*foo*⟩ addressable by descending twice to the left, per the associated address. The value of ⟨*baz*⟩ is addressable twice to the right, and ⟨*bar*⟩ is reached by the alternating path associated with it.

The semantics of the record delimiters is unspecified in cases of duplicate or overlapping addresses. In the current implementation, no exception is raised, but one field value may be overwritten by another partly or in full.

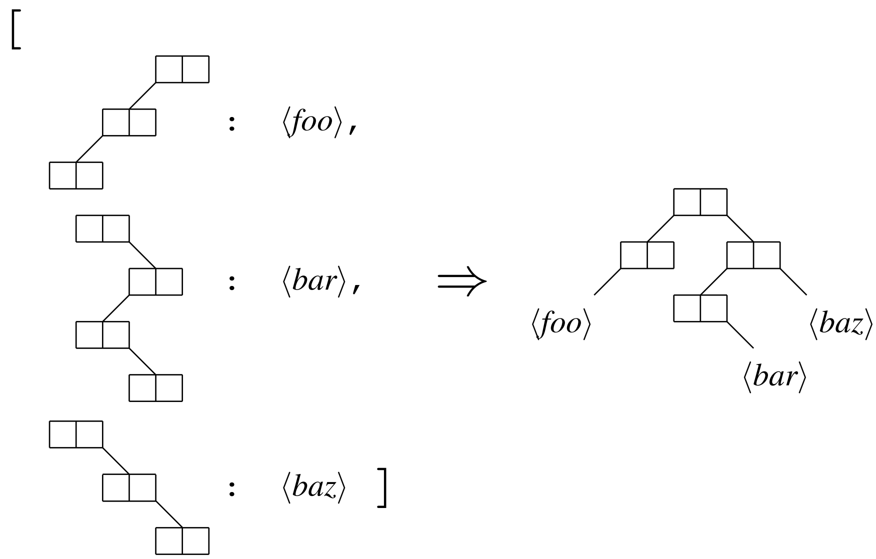


Figure 5.3: Record delimiters store the data at offsets relative to the root.

$$\begin{array}{ccccccc}
(& \langle \mathit{operand} \rangle & , & \langle \mathit{operand} \rangle & , & \cdots &) \\
\Updownarrow & \Updownarrow & \Updownarrow & \Updownarrow & \Updownarrow & & \Updownarrow \\
\overbrace{-[\langle \mathit{pretext} \rangle -[} & \overbrace{\langle \mathit{operand} \rangle} & \overbrace{]-\langle \mathit{intext} \rangle -[} & \overbrace{\langle \mathit{operand} \rangle} & \overbrace{]-\langle \mathit{intext} \rangle -[} & \cdots & \overbrace{]-\langle \mathit{postext} \rangle\,]-}
\end{array}$$

Figure 5.4: analogy between an expression with text delimiters and a tuple

-[]-- text delimiters

These operators follow a different pattern than the other data delimiters, because they don't enclose a comma separated sequence of operands. One way of understanding them is in syntactic terms according to the discussion of dash bracket notation on page 118. Alternatively, they can be viewed as delimiting operators forming an expression analogous to a tuple. The left parenthesis corresponds to something of the form -[⟨*pretext*⟩-[, the right parenthesis corresponds to]-⟨*postext*⟩]-, and the rôle of a comma is played by]-⟨*intext*⟩-[. This analogy is depicted in Figure 5.4.

- The embedded text can be arbitrarily long and can include line breaks, making the delimiters very thick operators, but operators nevertheless.
- In order for the expression to be well typed, the operands must evaluate to lists of character strings.
- Each of these operators has the semantic effect of concatenating its operands with the embedded text either before, between, or after the operands, as explained on page 118.
- The embedded text is not an operand but a hard coded feature of the operator. One might think in terms of a countable family of such operators, each induced by its respective embedded text.

5.2.2 Functional delimiters

The remaining aggregate operators from Table 5.8, represent functional combining forms. With the exception of -+...+-, they all pertain to conditional evaluation in some way. Although they normally enclose a comma separated sequence of operands, they can also be used with an empty sequence, as in -++-. In this form, the pair of operators together represent a function that applies to a list of operands rather than enclosing them. For example, -!p,q,r!- is semantically equivalent to -!!- <p,q,r>. The latter alternative is more useful in situations where the list of operands is generated at run time and can't be explicitly stated in the source.[1]

Composition

The simplest and most frequently used functional combining form is the composition operator, -+...+-, which denotes composition of a sequence of functions given by the expressions it encloses. That is, a composition of functions f_0 through f_n applied to an argument x evaluates to the nested application.

$$\texttt{-+}f_0\texttt{,}\,f_1\texttt{,}\ \dots f_n\texttt{+-}\ \ x \equiv f_0\ f_1\ \dots f_n\ x$$

where function application is right associative. The commas are necessary as separators, because the expressions for f_0 through f_n may contain operators of any precedence.

[1] difficult to motivate until you've had some practice at using higher order functions routinely

Composition example In a composition of functions, the last one in the sequence is necessarily evaluated first, as this example of a composition of three pointers shows.

```
$ fun --m="-+~&x,~&h,~&t+- <'foo','bar','baz'>" --c
'rab'
```

The tail of the list, `<'bar','baz'>` is computed first by `~&t`, then the head of the tail, `'bar'`, by `~&h`, and finally the reversal of that by `~&x`.

Optimization of composition Compositions are automatically optimized where possible. For example, the three functions in the above sequence can be reduced to two.

```
$ fun --main="-+~&x,~&h,~&t+-" --decompile
main = compose(reverse,field(0,(0,&)))
```

Optimizations may also affect the "eagerness" of a composition.

```
$ fun --m="-+constant'abc',~&t,~&h,~&x+-" --d
main = constant 'abc'
```

The constant function returns a fixed value regardless of its argument, so there is no need for the remaining functions in the composition to be retained.

Cumulative conditionals

The cumulative conditional form, `-?...?-`, is used to define a function by cases. Its normal usage follows this syntax.

```
-?
   ⟨predicate⟩:  ⟨function⟩,
        ⋮
   ⟨predicate⟩:  ⟨function⟩,
   ⟨default function⟩ ?-
```

The entire expression represents a single function to be applied to an argument.

- Each predicate in the sequence is applied to the argument in the order they're written, until one is satisfied.
- The function associated with the satisfied predicate is applied to the argument, and the result of that application is returned as the result of the whole function.
- The semantics is non-strict insofar as functions associated with unsatisfied predicates are not evaluated, nor are predicates or functions later in the sequence.
- If no predicate is satisfied, then the default function is evaluated and its result is returned.

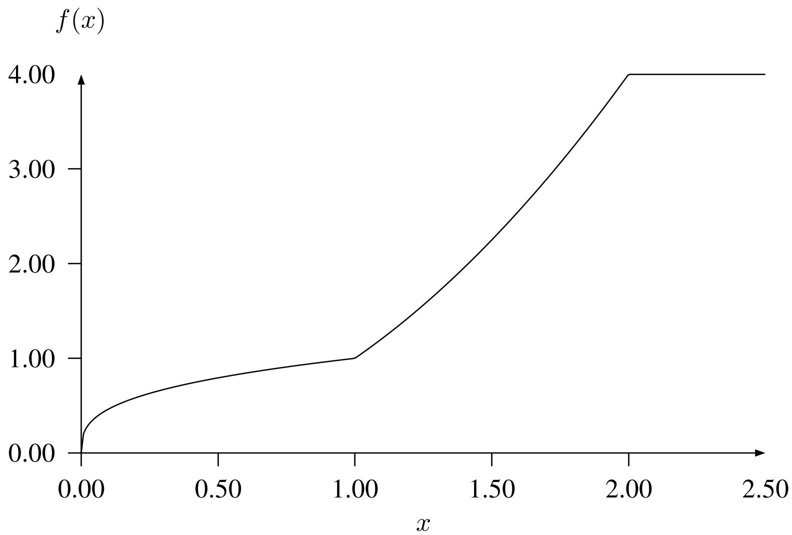


Figure 5.5: model of an inflationary cosmology according to f-theory

A simple contrived example of a function defined by cases is shown in Figure 5.5. The definition of this function is as follows.

$$f(x) = \begin{cases} 0 & \text{if} \quad x \leq 0 \\ \sqrt[3]{x} & \text{if} \quad 0 < x \leq 1 \\ x^2 & \text{if} \quad 1 < x \leq 2 \\ 4 & \text{otherwise} \end{cases}$$

This function can be expressed as shown using the `-?...?-` operators,

```
f = -?
        fleq\0.: 0.!,
        fleq\1.: math..cbrt,
        fleq\2.: math..mul+ ~&iiX,
        4.!?-
```

where `fleq` is defined as `math..islessequal`, the partial order relation on floating point numbers from the host system's C library, by way of the virtual machine's `math` library interface. The predicate `fleq\`k uses the reverse binary to unary combinator. When applied to an argument x it evaluates as `fleq\`k $x =$ `fleq` (x, k), which is true if $x \leq k$. The exclamation points represent the constant combinator.

Logical operators

The remaining aggregate operators in Table 5.8 support cumulative conjunction and two forms of cumulative disjunction. Similarly to the cumulative conditional, they all have a non-strict semantics, also known as short circuit evaluation.

- Cumulative conjunction is expressed in the form `-&`f_0`,`f_1`, ...`f_n`&-`. Each f_i is applied to the argument in the order they're written. If any f_i returns an empty value, then an empty value is the result, and the rest of the functions in the sequence aren't evaluated. If all of the functions return non-empty values, the value returned by last function in the sequence, f_n, is the result.
- Cumulative disjunction is expressed in the form `-|`f_0`,`f_1`, ...`f_n`|-`. Similarly to conjunction, each f_i is applied to the argument in sequence. However, the first non-empty value returned by an f_i is the result, and the remaining functions aren't evaluated. If every function returns an empty value, then an empty value is the result.
- An alternative form of cumulative disjunction is `-!`f_0`,`f_1`, ...`f_n`!-`. This form has a somewhat more efficient implementation than the one above, but will return only a `true` boolean value (`&`) rather than the actual result of a function f_i when it is non-empty, for $i < n$. This result is acceptable when the function is used as a predicate in a conditional form, because all non-empty values are logically equivalent.

Some examples of each of these combinators are the following.

```
$ fun --m="-&~&l,~&r&- (0,1)" --c
0
$ fun --m="-&~&l,~&r&- (1,2)" --c
2
$ fun --m="-|~&l,~&r|- (0,1)" --c
1
$ fun --m="-|~&l,~&r|- (1,2)" --c
1
$ fun --m="-!~&l,~&r!- (0,1)" --c
1
$ fun --m="-!~&l,~&r!- (1,2)" --c
&
```

Interpretation of exclamation points by the `bash` command line interpreter, even within a quoted string, can be suppressed only by executing the command `set +H` in advance, which is not shown.

5.2.3 Lifted delimiters

All of the aggregate operators in Table 5.8 follow a consistent convention regarding suffixes. The left operator of the pair (such as `<` or `{`) may be followed by arbitrarily many periods (as in `<.` or `{..`). For the text delimiters, the suffix is placed after the second opening dash bracket (as in `-[`⟨*text*⟩`-[.`). The closing operators (e.g., `>` and `}`) take no suffix.

The effect of a period in an aggregate operator suffix is best described as converting a data constructor to a functional combining form, with each subsequent period "lifting" the order by one. Periods used in functional combining forms such as `-|.` only lift their order. These concepts may be clarified by some illustrations.

First order list valued functions

The first order case is easiest to understand. The expression

$$\texttt{<}f_0\texttt{,}\, f_1\texttt{,}\,\dots f_n\texttt{>}$$

where each f_i is a function, represents a list of functions, but the expression

$$\texttt{<.}f_0\texttt{,}\, f_1\texttt{,}\,\dots f_n\texttt{>}$$

represents a function returning a list. When this function is applied to an argument x, the result is the list

$$\texttt{<}f_0\ x\texttt{,}\, f_1\ x\texttt{,}\,\dots f_n\ x\texttt{>}$$

That is, all functions are applied to the same argument, and a list of their results is made.

These distinctions are illustrated as follows. First we have a list of three trigonometric functions, which is each compiled to a virtual machine library function call.

```
$ fun --m="<math..sin,math..cos,math..tan>" --c %fL
<
   library('math','sin'),
   library('math','cos'),
   library('math','tan')>
```

The function returning the list of the results of these three functions is expressed with a suffix on the opening list delimiter.

```
$ fun --m="<.math..sin,math..cos,math..tan>" --c %f
couple(
   library('math','sin'),
   couple(
      library('math','cos'),
      couple(library('math','tan'),constant 0)))
```

This function constructs a structure following the representation shown in Figure 5.2. To evaluate the function, we can apply it to the argument of 1 radian.

```
$ fun --m="<.math..sin,math..cos,math..tan> 1." --c %eL
<8.414710e-01,5.403023e-01,1.557408e+00>
```

The result is a list of floating point numbers, each being the result of one of the trigonometric functions.

Text templates

The same technique can be used for rapid development of document templates in text processing applications.

```
$ fun --m="-[Dear -[. ~&iNC ]-,]- 'valued customer'" --show
Dear valued customer,
```

A first order function made from text delimiters, with functions returning lists of strings as the operands, can generate documents in any format from specifications of any type. In this example, the document is specified by a single character string, which need only be converted to a list of strings by the `~&iNC` pseudo-pointer.

Lifted functional combinators

A suffix on an opening aggregate operator such as $\texttt{-+}$ raises it to a higher order. A function of the form

$$\texttt{-+.}\ h_0\texttt{,}h_1\texttt{,}\ \dots h_n\ \texttt{+-}$$

applied to an argument u will result in the composition

$$\texttt{-+}\ h_0\ u\texttt{,}h_1\ u\texttt{,}\ \dots h_n\ u\ \texttt{+-}$$

If there are two periods, the function is of a higher order. When applied to an argument v, the result is a function that still needs to be applied to another argument to yield a first order functional composition.

$$\begin{aligned}(\texttt{-+..}\ h_0\texttt{,}h_1\texttt{,}\ \dots h_n\ \texttt{+-}\ v)\ u &\equiv \texttt{-+.}\ h_0\ v\texttt{,}h_1\ v\texttt{,}\ \dots h_n\ v\ \texttt{+-}\ u\\ &\equiv \texttt{-+}\ (h_0\ v)\ u\texttt{,}(h_1\ v)\ u\texttt{,}\ \dots (h_n\ v)\ u\ \texttt{+-}\end{aligned}$$

This pattern generalizes to any number of periods, although higher numbers are less common in practice. It also applies to other aggregate operators such as logical and record delimiters, but a more convenient mechanism for higher order records using the `$` operator is explained in the next chapter. Lambda abstraction using the `.` operator is another alternative also introduced subsequently.

Example Lifted functional combinators, like any higher order functions, are used mainly to abstract common patterns out of the code to simplify development and maintenance. One way of thinking about a lifted composition is as a mechanism for functional templates or wrappers.

A small but nearly plausible example is shown in Listing 5.1. Some language features used in this example are introduced in the next chapter, but the point relevant to the present discussion is the `wrapper` function.

The wrapper takes the form of a lifted composition

$$\texttt{-+.}\langle\textit{back end}\rangle\texttt{!,~\&,}\langle\textit{front end}\rangle\texttt{!+-}$$

where the exclamation points represent the constant functional combinator. When applied to any function f, the result will be the composition

$$\texttt{-+}\langle\textit{back end}\rangle\texttt{,}f\texttt{,}\langle\textit{front end}\rangle\texttt{+-}$$

wherein the front end serves as a preprocessor and the back end as a postprocessor to the function f.

Listing 5.1 when to use a higher order composition

```
#import std
#import nat

#library+

retype = # takes assignments of instance recognizers to type converters

-??-+ --<-[unrecognized type conversion]-!%>

promote = ..grow\100+ ..dbl2mp # 100 bits more precise than default 160

wrapper = # allows high precision for intermediate calculations

-+.
   retype<%EI: ..mp2dbl,%ELI: ..mp2dbl*,%ELLI: ..mp2dbl**>!,
   ~&,
   retype<%eI: promote,%eLI: promote*,%eLLI: promote**>!+-

rad_to_deg = # converts radians to degrees with high precision

wrapper mp..mul/1.8E2+ mp..div^/~& mp..pi+ mp..prec
```

In this example, the front end converts standard floating point numbers, vectors, or matrices thereof to arbitrary precision format. The function f is expected to operate on this representation, presumably for the sake of reduced roundoff error, and the final result is converted back to the original format.

The code in Listing 5.1, stored in a file named `promo.fun`, can be tested as follows.

```
$ fun promo.fun --archive
fun: writing `promo.avm'
$ fun promo --m="rad_to_deg 2." --c %e
1.145916e+02
```

A further point of interest in this example is the use of `-??-` as a function in the definition of `retype`. Effectively a new functional combining form is derived from the cumulative conditional, which takes a list of assignments of predicates to functions, but requires no default function. The predicates are meant to be type instance recognizers and the functions are meant to be type conversion functions.

```
$ fun promo --m="retype<%nI: mpfr..nat2mp> 153" --c %E
1.530E+02
```

A default function that raises an exception is supplied automatically because it is never meant to be reached.

```
$ fun promo --m="retype<%nI: mpfr..nat2mp> 'foo'" --c %E
fun:command-line: unrecognized type conversion
```

Listing 5.2 output from the command `$ fun --help outfix`

```
outfix operators
----------------
-?..?-  cumulative conditional with default case last
-+..+-  cumulative functional composition
-|..|-  cumulative ||, short circuit functional disjunction
-!..!-  cumulative !|, logical valued functional disjunction
-&..&-  cumulative &&, short circuit functional conjunction
 [..]   record delimiters
 <..>   list delimiters
 {..}   specifies sets as sorted lists with duplicates purged
 (..)   tuple delimiters
```

The content of the diagnostic message is the only feature specific to the definition of `retype` as a type converter.

5.3 Remarks

A quick summary of the aggregate operators described in this chapter is available interactively from the command

```
$ fun --help outfix
```

whose output is shown in Listing 5.2. Some of these, especially the logical operators, are comparable to infix operators that perform similar operations, as the listing implies and as the next chapter documents.

If you truly believe in the system of law you administer in my country, you must inflict upon me the severest penalty possible.

Ben Kingsley in *Gandhi*

6 Catalog of operators

With the previous chapter having exhausted what little there is to say about operators in general terms, this chapter details the semantics for each operator in the language on more of an individual basis. The operators are organized into groups roughly by related functionality, and ordered in some ways by increasing conceptual difficulty. An understanding of the conventions pertaining to arity and dyadic operators explained previously is a prerequisite to this chapter.

6.1 Data transformers

The six operators listed in Table 6.1 are used to express lists, assignments, sets, and trees, and some are already familiar from many previous examples. The set union operator, `|`, has only infix and solo arities, but the others have all four arities. These operators represent first order functions in their infix arities, and are dyadic in other arities (see Section 5.1.4). Hence, it is possible to write `t^:u` and `t^: u` interchangeably for a tree with root `t` and subtrees `u`.

	meaning	illustration		
`:`	list or assignment construction	`a:<b>`	≡	`<a,b>`
`^:`	tree construction	`r^:<v^:<>>`	≡	`~&V(r,<~&V(v,<>)>)`
`|`	union of sets	`{a,b}|{b,c}`	≡	`{a,b,c}`
`--`	concatenation of lists	`<a,b>--<c,d>`	≡	`<a,b,c,d>`
`-*`	left distribution	`a-*<b,c>`	≡	`<(a,b),(a,c)>`
`*-`	right distribution	`<a,b>*-c`	≡	`<(a,c),(b,c)>`

Table 6.1: data transformers

	meaning	illustration		
`!`	constant functional	`x! y`	$\equiv$	x
`/`	binary to unary combinator	`f/k x`	$\equiv$	`f(k,x)`
`\`	reverse binary to unary combinator	`f\k x`	$\equiv$	`f(x,k)`
`/*`	mapped binary to unary combinator	`f/*k <a,b>`	$\equiv$	`<f(k,a),f(k,b)>`
`\*`	mapped reverse binary to unary combinator	`f\*k <a,b>`	$\equiv$	`<f(a,k),f(b,k)>`

Table 6.2: constant forms

Consistently with the dyadic property, the infix and postfix forms of these operators have a higher order functional semantics. For example, `x--y` is a data value, the concatenation of a list `x` with a list `y`, but `--y` is the function that appends the list `y` to its argument, and `x--` is the function that appends its argument to `x`. In this way, the we have the required identity, `x--y` $\equiv$ `x-- y` $\equiv$ `--y x`, while the expressions `--y` and `x--` are also meaningful by themselves. A few more minor points are worth mentioning.

- The set union operator, `|`, is parsed as infix whenever it immediately follows an operand with no white space preceding it, and has an operand following it with or without white space. Otherwise it is parsed as a solo operator.
- The colon is considered to construct a list when used as an infix or solo operator, and an assignment when used as a prefix or postfix operator. Although the identity `a: b` $\equiv$ `a:b` $\equiv$ `:b a` is valid as far as concrete representations are concerned, only the equivalence between `a: b` and `:b a` is well typed (cf. Figures 5.1 and 5.2). On the other hand, typing is only a matter of programming style.
- As noted on page 59, the colon can also be used in pointer expressions pertaining to lists.
- The distribution operator `-*` in solo usage is equivalent to the pseudo-pointer `~&D` (page 75), and `*-` is equivalent to `~&rlDrlXS`.
- None of these operators has any suffixes.

6.2 Constant forms

The operators shown in Table 6.2 are normally used to express functions that may depend on hard coded constants. They have these algebraic properties.

- The constant combinator can be used either as a solo or as a postfix operator, and satisfies `! x` $\equiv$ `x!` for all `x`.
- The binary to unary combinators can be used as solo or infix operators, and are dyadic.

6.2.1 Semantics

The constant combinator and binary to unary combinators are well known features of functional languages, although the notation may vary.[1] The binary to unary combinators may also be familiar to C++ programmers as part of the standard template library.

Constant combinators

The constant combinator takes a constant operand and constructs a function that maps any argument to that operand. Such functions occur frequently as the default case of a conditional or the base case of a recursively defined function.

Binary to unary combinators

The binary to unary combinators / and \ take a function as their left operand and a constant as their right operand. The function is expected to be one whose argument is usually a pair of values. The combinator constructs a function that takes only a single value as an argument, and returns the result obtained by applying the original function to the pair made from that value along with the constant operand. For the / combinator, the constant becomes the left side of the argument to the function, and for the \ combinator, it becomes the right.

Standard examples are functions that add 1 to a number, `plus/1.` or `plus\1.`, and a function that subtracts 1 from a number, `minus\1.`. Normally the `plus` and `minus` functions perform addition or subtraction given a pair of numbers. In the latter case, the reverse binary to unary combinator is used specifically because subtraction is not commutative.

Currying A frequent idiomatic usage of the binary to unary combinator is in the expression `///`, which is parsed as `(/)/(/)`, and serves as a currying combinator. Any member f of a function space $(u \times v) \rightarrow w$ induces a function g in $u \rightarrow (v \rightarrow w)$ such that $g = \texttt{///}\ f$. This effect is a consequence of the semantics of these operators and their algebraic properties whose proof is a routine exercise.

Example The currying combinator allows any function that takes a pair of values to be converted to one that allows so-called partial application. For example, a partially valuable addition function would be `/// plus`. It takes a number as an argument and returns a function that adds that number to anything.

```
$ fun flo --m="((/// plus) 2.) 3." --c
5.000000e+00
```

The `plus` function is defined in the `flo` library distributed with the compiler.

[1] Curried functional languages don't need a binary to unary combinator, but the reverse binary to unary combinator could be a problem for them.

Mapped binary to unary combinators

The operators `/*` and `\*` serve a similar purpose to the binary to unary combinators above, but are appropriate for operations on lists. The left operand is a function taking a pair of values and the right operand is a constant, as above, but the resulting function takes a list of values rather than a single value. The constant operand is paired with each item in the list and the function is evaluated for each pair. A list of the results of these evaluations is returned.

This example uses the concatenation operator explained in the previous section to concatenate each item in a list of strings with an `'x'`.

```
$ fun --m="--\*'x' <'a','b','c'>" --c
<'ax','bx','cx'>
```

6.2.2 Suffixes

The binary to unary combinators `/` and `\` allow suffixes consisting of any sequence of the characters `$`, `|`, `;`, and `*`. that doesn't begin with `*`. The mapped binary to unary combinators `/*` and `\*` allow suffixes consisting of any sequence of the characters `$`, `=`, and `*`. Each character alters the semantics of the function constructed by the operator in a particular way. To summarize their effects briefly,

- the `$` makes the function apply to both sides of a pair
- the `|` makes the function triangulate over a list
- the `;` makes the function transform a list by deleting all items for which it is false
- the `*` makes the function apply to every item of a list
- the = flattens the resulting list of lists into the concatenation of its items.

When multiple characters are used in a single suffix, their effects apply cumulatively in the order the characters are written.

The suffix for `/` or `\` may not begin with `*` because in that case it is lexed as the `/*` or `\*` operator. However, the latter have the same semantics as the former would have if `*` could be used as the suffix. The triangulation and flattening suffixes are specific to the operators for which they are semantically more appropriate.

Examples

Some experimentation with these operator suffixes is a better investment of time than reading a more formal exposition would be. A few examples to get started are the following.

- This example shows how negative numbers can be removed from a list.

  ```
  $ fun flo --m="fleq/;0. <-2.,-1.,0.,1.,2.>" --c %eL
  <0.000000e+00,1.000000e+00,2.000000e+00>
  ```

	meaning	illustration		
`&`	pointer constructor	`&l`	≡	`(((),()),())`
`.`	composition or lambda abstraction	`~&h.&l`	≡	`~&hl`
`~`	deconstructor functional	`~p`	≡	`field p`
`:=`	assignment	`&l:=1! (2,3)`	≡	`(1,3)`

Table 6.3: pointer operations

- This examples shows the effect of a combination of list flattening and applying to both sides of a pair. Note the order of the suffixes.

```
$ fun --m="--\*=$'x' (<'a','b'>,<'c','d'>)" --c
('axbx','cxdx')
```

- This example shows a naive algorithm for constructing a series of powers of two.

```
$ fun --m="product/|2 <1,1,1,1,1>" --c %nL
<1,2,4,8,16>
```

The last example works because `f/|n <a,b,c,d>` is equivalent to

```
<a,f(n,b),f(n,f(n,c)),f(n,f(n,f(n,d)))>
```

Often there are several ways of expressing the same thing, and the choice is a matter of programming style. The function `product/|2` is equivalent to the pseudo-pointer `~&iNiCBK9` (see pages 76 and 87).

In case of any uncertainty about the semantics of these operators, there is always recourse to decompilation.

```
$ fun --m="--\*=$'x'" --decompile
main = fan compose(
   reduce(cat,0),
   map compose(cat,couple(field &,constant 'x')))
```

6.3 Pointer operations

A small classification of operators shown in Table 6.3 pertains to pointers in one way or another.

6.3.1 The ampersand

The ampersand has been used extensively in previous examples variously as the identity pointer, the true boolean value, or a notation for the pair of empty pairs, which are all equivalent in their concrete representations, but at this stage, it is best to think of it is as an operator.

The ampersand is an unusual operator insofar as it takes no operands and has only a solo arity. However, it allows a pointer expression as a suffix.

Although other operators employ pointer expressions in more specialized ways, the meaning of the ampersand operator is simply that of the pointer expression in its suffix. The semantics of pointer expressions is documented extensively in Chapter 2.

Most operators that allow pointer suffixes can accommodate pseudo-pointers as well, but the ampersand is meaningful only if its suffix is a pointer, except as noted below.

6.3.2 The tilde

The tilde operator can be used either as a prefix or as a solo operator. It has the algebraic property that `~ x` $\equiv$ `~x` for all `x`. A distinction is made nevertheless between the solo and the prefix usage because the latter has higher precedence.

The operand of the tilde operator can be any expression that evaluates to a pointer. A primitive form of such an expression would be a pointer specified by the ampersand operator, a field identifier from a record declaration, or a literal address from an a-tree or grid type. Tuples of these expressions are also meaningful as pointers, and the colon and dot operators can be used to build more pointer expressions from these.

The tilde operator is defined partly as a source level transformation that lets it depend on the concrete syntax of its operand. Pseudo-pointer suffixes for the ampersand operator, while not normally meaningful in themselves, are acceptable when the ampersand forms part of the operand of a tilde operator. The tilde in this case effectively disregards the ampersand and makes direct use of the pseudo-pointer suffix.

The result returned by the tilde operator is a either a virtual code program of the form `field` p for an pointer operand p, or a function of unrestricted form if its operand is a pseudo-pointer. The `field` combinator pertains to deconstructors, which are functions that return some part of their argument specified by a pointer.

6.3.3 Assignment

The assignment operator, `:=`, performs an inverse operation to deconstruction. It satisfies the equivalence

`~a a:=f x` $\equiv$ `f x`

for any address `a`, function `f`, and data `x`. It is also dyadic in all arities. Intuitively this relationship means that whereas deconstruction retrieves the value from a field in a structure, assignment stores a value in it.

Fields in the result that aren't specifically assigned by this operation inherit their values from the argument `x`. If `b` were an address different from `a`, then `~b a:=f x` would be the same as `~b x`. This condition defies a simple rigorous characterization, but the following examples should make it clear.

Usage

The address in an expression `a:=f x` can refer to a single field or a tuple of fields in the argument `x`. In the latter case, the function `f` should return a tuple of a compatible form.[2]

```
$ fun --m="&h:='c'! <'a','b'>" --c %sL
<'c','b'>
$ fun --m="(&h,&th):=~&thPhX <'a','b'>" --c %sL
<'b','a'>
```

- As the second example above shows, multiple fields can be referenced or interchanged by an assignment without interference, provided their destinations don't overlap.
- The address in an assignment can be a pointer expression containing constructors, (e.g., `&hthPX` instead of `(&h,&th)`), but it must be a pointer rather than a pseudo-pointer. (See Chapter 2 for an explanation.)
- If the address of an assignment refers to multiple fields and the function returns a value with not enough (such as an empty value) an exception is raised with the diagnostic message of "`invalid assignment`".

Suffixes

An optional pointer expression *s* may be supplied as a suffix, with the syntax `:=`*s*. The suffix can be a pointer or a pseudo-pointer, but it must be given by a literal pointer constant rather than a symbolic name.

The suffix is distinct from the operands and may be used in any arity. However, when a suffix is used in the prefix or infix arities, as in `:=`*s*`f` or `a:=`*s*`f`, and the right operand `f` begins with alphabetic character, `f` must be parenthesized to distinguish it from a suffix. In fact, any right operand to an assignment with or without a suffix must be parenthesized if it begins with an alphabetic character.

The purpose of the suffix is to specify a postprocessor. An expression `a:=`*s* `f` with a suffix *s* is equivalent to `-+~&`*s*`,a:=f+-` or `~&`*s*`+ a:=f`. This feature is a matter of convenience because assignments are almost always composed with deconstructors or pseudo-pointers in practice, as a regular user of the language will discover.

Non-mutability

The idea of storage is non-mutable as always. If `x` represents a store, then `a:=f` is a function that returns a new store differing from `x` at location `a`. Evaluating this function has no effect on the interpretation of `x` itself, as this example shows.

```
$ fun --m="x=<1> y=(&h:=2! x) z=(x,y)" --c %nLW,z
(<1>,<2>)
```

[2]If you're trying these examples, be sure to execute `set +H` first to suppress interpretation of the exclamation point by the `bash` command line interpreter.

The original value of x is retained in z despite the definition of y as x with a reassigned head.

Growing a new field

In order for the above equivalence to hold without exception, assignment to a field that doesn't exist in the argument causes it to grow one rather than causing an invalid deconstruction. For example, an attempt to retrieve the head of the tail of a list with only one item causes an invalid deconstruction, as expected,

```
$ fun --m="~&th <1>" --c %n
fun:command-line: invalid deconstruction
```

but retrieving that of a list in which it has been assigned doesn't.

```
$ fun --m="~&th &th:=2! <1>" --c %n
2
```

The assignment to the second position in the list either overwrites the item stored there if it exists (in a non-mutable sense) or creates a new one if it doesn't.

```
$ fun --m="&th:=2! <1>" --c %nL
<1,2>
```

It could also happen that other fields need to be created in order to reach the one being assigned. In that case, the new fields are filled with empty values.

```
$ fun --m="&tth:=2! <1>" --c %nL
<1,0,2>
```

It is the user's responsibility to ensure that fields created in this way are semantically meaningful and well typed.

```
$ fun --m="&tth:=2.! <1.>" --c %eL
fun: writing `core'
warning: can't display as indicated type; core dumped
```

An empty value is not well typed in a list of floating point numbers.

Manual override

Assignment can be used to override the usual initialization function for a record and set the value of a field "by hand". (See Section 4.2.3 for more about initialization functions in records.) A simple illustration is a record r with two natural type fields u and w, wherein w is meant track the value of u and double it.

```
r :: u %n w %n ~u.&NiC
```

By default, this mechanism works as expected.

```
$ fun --m="r :: u %n w %n ~u.&NiC x= _r%P r[u: 1]" --s
r[u: 1,w: 2]
```

However, if `u` is reassigned, the initialization function is bypassed, and `w` retains the same value.

```
$ fun --m="r::u %n w %n ~u.&NiC x=_r%P u:=3! r[u: 1]" --s
r[u: 3,w: 2]
```

Obviously, invariants meant to be maintained by the record specification can be violated by this technique, so it is used only as a matter of judgment when circumstances warrant. The normal way of expressing functions returning records is with the `$` operator, explained subsequently in this chapter, which properly involves the initialization functions.

Changing a field in a record by an assignment can also cause it to be badly typed. Even if the field itself is changed to an appropriate type, the type instance recognizer of a record takes the invariants into account.

```
$ fun --m="r::u %n w %n ~u.&NiC x=_r%I u:=3! r[u: 1]" -c %b
false
```

For this reason, the updated record will not be cast to the type `_r`.

```
$ fun --m="r::u %n w %n ~u.&NiC x= u:=3! r[u: 1]" --c _r
fun: writing 'core'
warning: can't display as indicated type; core dumped
```

The badly typed record was displayable in previous examples only by the `_r%P` function, which doesn't check the validity of its argument.

6.3.4 The dot

The dot operator has two unrelated meanings, one for relative addressing, making it topical for this section, and the other for lambda abstraction. The operator allows either an infix or a postfix arity. The infix usage pertains to relative addressing, and the postfix usage to lambda abstraction.

Relative addressing

An expression of the form `a.b` with pointers `a` and `b` describes the address `b` relative to `a`. Semantically the dot operator is equivalent to the `P` pointer constructor (pages 63 and 79), but the latter appears only in literal pointer constants, whereas the dot operator accommodates arbitrary expressions involving literal or symbolic names.

In many cases, the deconstruction of a value `x` by a relative address `~a.b` could also be accomplished by first extracting the field `a` and then the field `b` from it, as in `~b ~a x`. In these cases, the dot notation serves only as a more concise and readable alternative, particularly for record field identifiers (see page 154 for an example).

The equivalence between `~a.b x` and `~b ~a x` holds when `a` is a pseudo-pointer, a pointer referring to only a single field, or a pointer equivalent to the identity, such as `&lrX`, `&C`, `&nmA`, or `&V`. However, an interpretation more in keeping with the intuition of relative addressing is applicable when the left operand, `a`, represents a pointer to multiple fields. In this case, the pointer `b` is relative to each of the fields described by `a`, and the above mentioned equivalence doesn't hold.

Pointers to multiple fields are expressions like `&b`, `&hthPX`, or a pair of field identifiers `(foo,bar)`. The dot operator could be put to use in taking the `bar` field from the first two records in a list by `&hthPX.bar`.

Lambda abstraction

An alternative to the use of combinators to specify functions is by lambda abstraction, so called because its traditional notation is $\lambda x.\ f(x)$, where x is a dummy variable and $f(x)$ is an expression involving x. This idea has a well established body of theory and convention, to which the current language adheres for the most part. However, the λ symbol itself is omitted, because the dot as a postfix operator is sufficiently unambiguous, and dummy variables are enclosed in double quotes to distinguish them from identifiers.

Parsing The postfix arity of the dot operator is indicated when it is immediately preceded by an operand and followed by white space, which is then followed by another operand. This last condition is necessary because lambda abstraction is mainly a source level transformation.

When it is used for lambda abstraction, the dot operator has a lower precedence than function application and any non-aggregate operator except declarations (= and `::`). It is also right associative. These conditions imply the standard convention that the body of an abstraction extends to the end of the expression or to the next enclosing parenthesis, comma, or other aggregate operator.

Semantics The function defined by a lambda abstraction `"x".` $f($`"x"`$)$ is computed by substituting the argument to the function for all free occurrences of `"x"` in the expression $f($`"x"`$)$ and evaluating the expression.

Free occurrences of a variable in the body of a lambda abstraction are usually all occurrences except in contrived examples to the contrary. Technically a free occurrence of a variable `"x"` is one that doesn't appear in any part of a nested lambda abstraction expressed in terms of a variable with the same name (i.e., another `"x"`).

An example of an occurrence that isn't a free occurrence of `"x"` is in the expression `"x". "x". "x"`. This expression nevertheless has a well defined meaning, which is the constant function returning the identity function, `~&!`.[3] Nested lambda abstractions are ordinarily an elegant specification method for higher order functions that can be more easily readable than the equivalent combinatoric form.

[3]With no opportunity for substitution, applying this expression to any argument yields `"x". "x"`, which is the identity function because applying it to any argument yields the argument.

	meaning	illustration		
`->`	iteration	`p->f`	≡	`p?(p->f+ f,~&)`
`^=`	fixed point computation	`f^= x`	≡	`f^= f x`
`+`	composition	`f+g x`	≡	`f g x`
`;`	reverse composition	`g;f x`	≡	`f g x`
`@`	composition with a pointer	`g@h`	≡	`g+~&h`

Table 6.4: sequencing operators

Pattern matching Lambda abstractions can also be expressed in terms of lists or tuples of dummy variables, in any combination and nested to any depth. The syntax for lists and tuples of dummy variables is the same as usual, namely a comma separated sequence enclosed by angle brackets or parentheses.

The reason for using a pair of dummy variables would be to express a function that takes a pair of values as an argument and needs to refer to each value individually. When a pair of dummy variables is used, each component of the argument is identified with a distinct variable, and they can appear separately in the expression. For example, a function that concatenates a pair of lists in the reverse order could be expressed as

```
("x","y"). "y"--"x"
```

When a function is defined as a lambda abstraction with a tuple of dummy variables, it should be applied only to arguments that are tuples with at least as many components, or else an exception may be raised due to an invalid deconstruction. Similarly, a list of dummy variables in the definition means that the function should be applied only to lists with at least one item for each dummy variable. For nested lists or tuples, each component of the argument should match the arity or length of the corresponding component in the nested list or tuple of dummy variables. See page 164 for a related discussion.

Repeating a dummy variable within the same pattern, as in `("x","x"). "x"`, is allowed but has no special significance.[4] There is nothing to compel this function to be applied only to pairs of equal values. The component of the argument to which a repeated dummy variable refers in the body of the abstraction is unspecified. Note that this example differs from the case of a nested lambda abstraction, wherein repeated variables have a standard interpretation as discussed above.

6.4 Sequencing operations

Five operators pertain feeding the output from one function into another or feeding it back to the same one. They are listed in Table 6.4. There are two for iteration and three for composition.

[4] An alternative semantics considered and rejected in the design of Ursala would allow a pattern with repetitions to express a partial function restricted to a domain matching the pattern. This semantics would be useful only in the context of a function defined by cases via multiple partial functions, which raises various practical and theoretical issues.

6.4.1 Algebraic properties

These operators are designed with various algebraic properties to be as convenient as possible in typical usage.

- The iteration combinator `->` allows all four arities and is fully dyadic.
- The fixed point iterator has postfix and solo arities, and satisfies `f^=` $\equiv$ `^= f`.
- The composition with pointers operator, `@`, has only postfix and solo arities, with the same algebraic properties as the fixed point iterator.
- The composition operator, `+`, lacks a prefix arity but is otherwise dyadic.
- The reverse composition operator, `;`, also lacks a prefix arity. It is postfix dyadic, but its solo arity satisfies `(; f) g` $\equiv$ `f; g`.

The pointer s in f`@`s is a suffix rather than an operand, and must be a literal pointer constant rather than an identifier or expression. Without a suffix, the identity pointer is inferred, which has no effect. A late addition to the language, this operator's purpose is more to reduce the clutter in many expressions than to provide any more functionality.

6.4.2 Semantics

The semantics of these operators are as simple as they look, and require no lengthy discourse.

- The fixed point iterator, `^=`, applies a function to the original argument, then applies the function again to the result, and so on, until two consecutive results are equal. The last result obtained is the one returned. Non-termination is a possibility.[5]
- The iteration combinator in a function `p->f` similarly applies the function `f` repeatedly, but uses a different stopping criterion. The predicate `p` is applied to each result from `f`, and the first result for which `p` is false is returned. The result may also be the original argument if `p` isn't satisfied by it, in which case `f` is never evaluated.
- The composition operator in a function `f+g` applies `g` to the argument, feeds the output from `g` into `f`, and returns the result from `f`. This function is the infix equivalent of one given by the aggregate operator `-+f,g+-`.
- The reverse composition operator, used in a function `f;g`, is semantically equivalent to the composition operator with the operands interchanged, i.e., `g+f` or `-+g,f+-`.

[5] See page 78 for a discussion of equality.

6.4.3 Suffixes

All of the operators in Table 6.4 can be used with a suffix. The suffix can be used in any arity the operators allow. There are three different conventions followed be these operators regarding suffixes.

- The iterations `->` and `^=` allow a literal pointer constant as a suffix.
- The fixed point iterator `^=` also allows the = character in a suffix.
- The composition operators + and `;` can take a suffix consisting of any sequence of the characters `*`, =, `.`, and `$`.

Iteration postprocessors

A pointer constant s serves as a postprocessor to the iteration operators, similarly to its use by the assignment operator. That is, `p->`s`f` is equivalent to `~&`s`+ p->f`, and `f^=`s is equivalent to `~&`s`+ f^=`. The right operand to `->` in its infix or prefix arities must be parenthesized to distinguish it from a suffix if it begins with an alphabetic character.

For the fixed point iterator `^=`, a suffix of = can be used, as in `^==`, either with or without a pointer constant. The effect of the = is to generalize the stopping criterion to compare each newly computed result with every previous result, rather than comparing it only to its immediate predecessor. This criterion makes the computation more costly both in time and memory usage, but will allow it to terminate in cases of oscillation, where the alternative wouldn't.

Embellishments to composition

The suffixes to the composition operators alter the semantics of the function they would normally construct in the following ways.

- The `*` makes the function apply to all items of a list.
- The = composes the function with a list flattening postprocessor.
- The `$` makes the function apply to both sides of a pair.
- The `.` makes the function transform a list by deleting the items that falsify it.

These explanations may be supplemented by some examples.

```
$ fun --m="~&h+*~&t <'ab','cd','ef','gh'>" --c
'bdfh'
$ fun --m="~&t+=~&t <'ab','cd','ef','gh'>" --c
'efgh'
$ fun --m="~&h+$~&t (<'ab','cd'>,<'ef','gh'>)" --c
('cd','gh')
$ fun --m="~&t+.~&t <'abc','de','fgh','ij'>" --c
<'abc','fgh'>
```

	meaning	illustration		
`?`	conditional	`~&w?(~&x,~&r)`	≡	`~&wxrQ`
`^?`	recursive conditional	`p^?(f,g)`	≡	`refer p?(f,g)`
`?=`	comparing conditional	`x?=(f,g)`	≡	`~&==x?(f,g)`
`?<`	inclusion conditional	`x?<(f,g)`	≡	`~&-=x?(f,g)`
`?$`	prefix conditional	`x?$(f,g)`	≡	`~&=]x?(f,g)`

Table 6.5: conditional forms

The functions above are equivalent to the pseudo-pointers `~&thPS`, `~&ttL`, `~&bth`, and `~&ttPF`. When multiple characters appear in the same suffix, their effect is cumulative and the order matters.

```
$ fun --m="~&t+.=~&t <'abc','de','fgh','ij'>" --c
'abcfgh'
$ fun --m="~&t+.=~&t" --decompile
main = compose(reduce(cat,0),filter field(0,(0,&)))
```

6.5 Conditional forms

Several forms of non-strict evaluation of functions conditioned on a predicate are afforded by the operators listed in Table 6.5. These operators have only postfix and solo arities, and therefore are not dyadic, but they share the algebraic property

$$\texttt{(p?)(f,g)} \equiv \texttt{(?)(p,f,g)}$$

where these expressions are fully parenthesized to emphasize the arity. More frequent idiomatic usages are `p?/f g` and `?(p,~&/f g)`, *etcetera*, with line breaks per stylistic convention.

6.5.1 Semantics

These operators are defined in terms of the virtual machine's `conditional` combinator, a second order function that takes a predicate p and two functions f and g to a function that evaluates to f or g depending on the predicate.

$$\texttt{conditional(}p\texttt{,}f\texttt{,}g\texttt{)}\;\; x = \begin{cases} f\,\texttt{(}x\texttt{)} & \text{if}\;\; p\,\texttt{(}x\texttt{)}\;\; \text{is non-empty} \\ g\,\texttt{(}x\texttt{)} & \text{otherwise} \end{cases}$$

The non-strict semantics means the function not chosen is not evaluated and therefore unable to raise an exception. This behavior is similar to the `if...then...else` statement found in most languages.

- The `?` operator in a function `p?(f,g)` directly corresponds to the `conditional` combinator with a predicate `p` and functions `f` and `g`.

- The `?=` operator in a function `x?=(f,g)` allows any arbitrary constant `x` in place of a predicate, and translates to the `conditional` combinator with a predicate that tests the argument for equality with the constant.[6]
- The `?$` operator in a function `x?$(f,g)` allows any list or string constant `x` in place of a predicate, and translates to the `conditional` combinator with a predicate that holds for any list or string argument having a prefix of `x`.
- The `?<` operator in a function `x?<(f,g)` with a constant list or set `x` tests the argument for membership in `x` rather than equality.
- The `^?` operator in a function `p^?(f,g)` translates to a `conditional` wrapped in a `refer` combinator, equivalent to `refer conditional(p,f,g)`.

The `refer` combinator is used in recursively defined functions. An expression of the form `(refer f) x` evaluates to `f ~&J(f,x)`. See pages 44 and 72 for further explanations.

6.5.2 Suffixes

The conditional operators listed in Table 6.5 all allow pointer expressions as suffixes, and the `^?` additionally allows suffixes containing the characters =, `$`, and <.

Equality and membership suffixes

The `^?` operator with a suffix = is a recursive form of the `?=` operator. That is, the function `p^?=(f,g)` is equivalent to `refer p?=(f,g)`. Similarly, `p^?<(f,g)` is equivalent to the function `refer p?<(f,g)`, and `p^?$(f,g)` is equivalent to the function `refer p?$(f,g)`. The =, `$` and < characters are mutually exclusive in a suffix. The effect of using more than one together is unspecified.

Pointer suffixes

The pointer expression s in a function `p?`s`(f,g)` serves as a preprocessor to the predicate `p`, making the function equivalent to `(p+ ~&`s`)?(f,g)`. The expression s can be a pseudo-pointer but must be a literal constant. Note that only the predicate `p` is composed with `~&`s, not the functions `f` and `g`.

For the `?=` and `?<` operators, the pointer expression is composed with the implied predicate. Hence, `x?=`s`(f,g)` is equivalent to `(~&E/x+ ~&`s`)?(f,g)` and `x?<`s`(f,g)` is equivalent to `(~&w\x+ ~&`s`)?(f,g)`. (See page 78 for a reminder about the equality and membership pseudo-pointers `E` and `w`.)

Combined suffixes

A pointer expression and one of < or = may be used together in the same suffix of the `^?` operator, as in `p^?=`s`(f,g)` or `p^?<`s`(f,g)`, with the obvious interpretation as a recursive form of one of the above operators with a pointer suffix.

[6] see page 78 for a discussion of equality

<table>
<tr><th></th><th>meaning</th><th>illustration</th><th></th><th></th></tr>
<tr><td><code>&&</code></td><td>conjunction</td><td><code>f&&g</code></td><td>≡</td><td><code>f?(g,0!)</code></td></tr>
<tr><td><code>||</code></td><td>semantic disjunction</td><td><code>f||g</code></td><td>≡</td><td><code>f?(f,g)</code></td></tr>
<tr><td><code>!|</code></td><td>logical disjunction</td><td><code>f!|g</code></td><td>≡</td><td><code>f?(&!,g)</code></td></tr>
<tr><td><code>^&</code></td><td>recursive conjunction</td><td><code>f^&g</code></td><td>≡</td><td><code>refer f&&g</code></td></tr>
<tr><td><code>^!</code></td><td>recursive disjunction</td><td><code>f^!g</code></td><td>≡</td><td><code>refer f!|g</code></td></tr>
<tr><td><code>-=</code></td><td>membership</td><td><code>f-= s</code></td><td>≡</td><td><code>~&w^(f,s!)</code></td></tr>
<tr><td><code>==</code></td><td>comparison</td><td><code>f== x</code></td><td>≡</td><td><code>~&E^(f,x!)</code></td></tr>
<tr><td><code>~<</code></td><td>non-membership</td><td><code>f~< s</code></td><td>≡</td><td><code>^wZ(f,s!)</code></td></tr>
<tr><td><code>~=</code></td><td>inequality</td><td><code>f~= x</code></td><td>≡</td><td><code>^EZ(f,x!)</code></td></tr>
</table>

Table 6.6: predicate combinators

6.6 Predicate combinators

A selection of operators for constructing predicates useful for conditional forms among other things is shown in Table 6.6. There are operators for testing of equality and membership in normal and negated forms, and for several kinds of functional conjunction and disjunction.

6.6.1 Boolean operators

The boolean operators in Table 6.6 are `&&`, `||`, `!|`, `^&`, and `^!`. Algebraically, they allow all four arities and are fully dyadic. Semantically, they are second order functions that take functions rather than data values as their operands, and their results are functions. The functions they return have a non-strict semantics. There are currently no suffixes defined for these operators.

Non-strictness

The non-strict semantics means that in their infix usages, the right operand isn't evaluated in cases where the logical value of the result is determined by the left. A prefix usage such as `&&q` represents a function that needs to be applied to a predicate `p`, and will then construct a predicate equivalent to the infix form `p&&q`. The resulting predicate therefore evaluates `p` first and then `q` only if necessary. Similar conventions apply to other arities.

Semantics

The meanings of these operators can be summarized as follows.

- A function `f&&g` applies `f` to the argument, and returns an empty value iff the result from `f` is empty, but otherwise returns the result obtained by applying `g` to the argument.

- A function `f||g` applies `f` to the argument, and returns the result from `f` if it is non-empty, but otherwise returns the result of applying `g` to the argument. Although it is semantically equivalent to `f?(f,g)`, it is usually more efficient due to code optimization.
- A function `f!|g` is similar to `f||g` but even more efficient in some cases. It will return a true boolean value `&` if the result from `f` is non-empty, but otherwise will return the result from `g`.
- The function `f^&g` is equivalent to `refer f&&g`.
- The function `f^!g` is equivalent to `refer f!|g`.

The `refer` combinator is used in recursively defined functions. An expression of the form `(refer f) x` evaluates to `f ~&J(f,x)`. See pages 44 and 72 for further explanations.

The aggregate operators `-&f,g&-`, `-|f,g|-`, and `-!f,g!-` have a similar semantics to the first three of these operators but allow arbitrarily many operands. See page 192 for more information.

6.6.2 Comparison and membership operators

The operators `==`, `~=`, `-=`, and `~<` from Table 6.6 pertain respectively to equality, inequality, membership, and non-membership. These operators have no suffixes. They allow all four arities but are dyadic only in their postfix arity. For their prefix arities, they share the algebraic property

`f; ==x` $\equiv$ `f==x`

but in their solo arities they are only first order functions taking pairs of data to boolean values.

- In the infix usage, these operators are second order functions that require a function as a left operand and a constant as the right operand. They construct a function that works by applying the given function to the argument and testing its return value against the given constant, whether for equality, inequality, membership, or non-membership, depending on the operator.
- In the prefix usage, the operand is a constant and the result is a function that tests its argument against the constant.
- In the postfix usage `f==`, as implied by the dyadic property, a function `f` as an operand induces a function that can be applied to a constant `x`, to obtain an equivalent function to `f==x`, and similarly for the other three operators.

For the membership operators, the constant or the right operand should be a set or a list, and the result from the function if any should be a possible member of it. For example, `-='0123456789'` is the function that tests whether its argument is a numeric character, and returns a true value if it is.

<table>
<tr><th></th><th>meaning</th><th>illustration</th><th></th><th></th></tr>
<tr><td>-</td><td>table lookup</td><td><'a': x,'b': y>-a</td><td>≡</td><td>x</td></tr>
<tr><td>..</td><td>library combinator</td><td>l..f</td><td>≡</td><td>library('l','f')</td></tr>
<tr><td>.|</td><td>run-time library replacement</td><td>lib.|func f</td><td>≡</td><td>f</td></tr>
<tr><td>.!</td><td>compile-time library replacement</td><td>lib.!func f</td><td>≡</td><td>f</td></tr>
</table>

Table 6.7: module dereferencing

6.7 Module dereferencing

Four operators shown in Table 6.7 are useful for access and control of library functions. Library functions can be those that are implemented in other languages and linked into the virtual machine such as the linear algebra and floating point math libraries, or they can be implemented in virtual code stored in `.avm` library files that are user defined or packaged with the compiler. The dash operator, `-`, is useful for the latter and the other operators are useful for the former.

6.7.1 The dash

This operator allows only an infix arity and has a higher precedence than most other operators. The left operand should be of a type t`%m` for some type t, which is to say a list of assignments of strings to instances of t, and the right operand must be an identifier.

Syntax

The dash operator is implemented partly as a source level transformation that allows it to have an unusual syntax. The identifier that is its right operand need not be bound to a value by a declaration elsewhere in the source. Rather, it should be identical to some string associated with an item of the left operand. The value of an expression `foo-bar` is the value associated with the string `'bar'` in the list `foo`. Although `'bar'` is a string, it is not quoted when used as the right operand to a dash operator.

- If the right operand to a dash operator is anything other than a single identifier, an exception is raised with the diagnostic message of "`misused dash operator`" during compilation.
- If the right operand s doesn't match any of the names in the left operand, an exception is raised with the message of "`unrecognized identifier:` s".

Semantics

Although it is valid to write a dash operator with a literal list of assignments of strings to values as its left operand

$$\texttt{<'}s_0\texttt{': }x_0\texttt{, ... '}s_n\texttt{': }x_n\texttt{>-}s_k$$

a more useful application is to have a symbolic name as the left operand representing a previously compiled library module.

Any source text containing `#library+` directives generates a library file with a suffix of `.avm` when compiled, that can be mentioned on the command line during a subsequent compilation. Doing so causes the name of the file (without the `.avm` suffix) to be available as a predeclared identifier whose value is the list of assignments of strings to values declared in the library. A usage like `lib-symbol` allows an externally compiled symbol from a library named `lib.avm` to be used locally, provided that file name is mentioned on the command line during compilation.

The `#import` directive serves a related purpose by causing all symbols defined in a library to be accessible as if they were locally declared. However, the dash operator is helpful when an external symbol has the same name as a locally declared symbol, because it provides a mechanism to distinguish them.

Type expressions

Type expressions associated with record declarations in modules are handled specially by the dash operator. The compiler uses a compressed format for type expressions to save space when storing them in library files. The dash operator takes this format into account.

When any identifier beginning with an underscore is used as the right operand to a dash operator, and its value is detected to be that of a compressed type expression, the value is uncompressed automatically. This effect is normally not noticeable unless the module containing a type expression is accessed by other means than the dash operator in an application that makes direct use of type expressions.

Compressed libraries

If a file containing `#library+` directives is compiled with the `--archive` command line option, the file is written in a compressed format. This compression is optional and is orthogonal to that of type expressions mentioned above.

The dash operator automatically detects whether its left operand is a compressed module and accesses it transparently. Operating on compressed modules otherwise requires uncompressing them explicitly, which can be performed by the function `%QI`. See page 132 for an example.

6.7.2 Library invocation operators

The other kind of library functions are those that are written in C or Fortran and are invoked directly by the virtual machine. The virtual machine code for a call to this kind of library function is essentially a stub

`library(`⟨*library name*⟩`,`⟨*function name*⟩`)`

containing the name of the library and the function as character strings, which are looked up at run time by an interpreter. The available libraries and function names are site specific, but can be viewed by executing the shell command

```
$ fun --help library
```

as shown in Listing 1.10 on page 46, and as documented in the `avram` reference manual.

Aside from invoking a library function by the `library` combinator explicitly as shown above, there are three operators intended to make it more convenient as shown in Table 6.7, which are the `..` (elipses), `.!`, and `.|` operators.

Syntax

Algebraically the library name is the left operand and the function name is the suffix for each of these operators. The right operand, if any, can be any expression representing a function. All three operators allow solo and postfix usage. The `.!` and `.|` operators allow infix usage and are postfix dyadic.

Syntactically the library name must be an identifier, which needn't be declared anywhere else because it is literally translated to a string by a source transformation, similarly to the right operand of a dash operator as explained above. Anything other than an identifier as the left operand to one of these operators causes a compile time exception.

The function name in the suffix may contain digits, which are not normally valid in identifiers, as well as letters and underscores.

Both the library and function names can be recognizably truncated or even omitted where there is no ambiguity (either because a function names is unique across libraries, or because a library has only one function).

Semantics

The operators differ in their semantics, as explained below.

The elipses The `..` allows only a postfix or solo arity, with the solo arity corresponding to the case where the library name is omitted. It is translated directly to the `library` combinator mentioned above with an attempt to complete any truncated library or function names at compile time.

- If there isn't a unique match found for either the library or the function name in the postfix usage `lib..func`, it is taken literally (even if no such function or library exists on the compile time platform).
- If there isn't a unique match found for the function name in the solo usage (i.e., with the library name omitted), then a compile time exception is raised with the diagnostic message "`unrecognized library function`".

Compile time replacement Integration of compatible replacements for external library functions is important for portability, but the library function is preferable where available for reasons of performance. The `.!` operator provides a way for a replacement function to be used in place of an unavailable library function. The determination of availability is made at compile time based on the virtual machine configuration on the compilation platform.

- An expression of the form `lib.!func f` evaluates to `f` if no unique match to the library function is found, but it evaluates to `lib..func` otherwise.
- A solo usage of the form `.!func f` behaves analogously, but obviously may fail to find a unique match for the library function in some cases where the usage above would not.
- Consistently with the dyadic property and solo semantics, an expression `.!func` or `lib.!func` by itself evaluates either to the identity function or to a constant function returning `lib..func`, depending on whether a matching library function is found during compilation.
- In any case, no compile time exception is raised, but run time errors are possible if a library function present on the compile time platform is absent from the target.

Run time replacement The `.|` operator provides a way for a replacement function to be used in place of an unavailable library function with the determination of availability made at run time.

- An expression of the form `lib.|func f` represents a function that performs a run time check for the availability of a function named `func` in a library named `lib`. If such a function exists and is unique, it is applied to the argument, but otherwise the function `f` is applied to the argument.
- A solo usage of the form `.|func f` behaves analogously, but searches every virtual machine library for a function named `func`.
- Consistently with the above usages, an expression `.|func` or `lib.|func` by itself represents a higher order function that needs to be applied to a function `f` in order to yield a meaningful combination of `lib..func` and `f`.
- This operator is unlikely to cause either compile time or run time errors, and will generate code that makes the best use of available library functions on the target in exchange for a slight run time overhead.

6.8 Recursion combinators

Four operators shown in Table 6.8 are grouped together loosely on the basis that they abstract common patterns of recursion, particularly over lists and trees.

	meaning	illustration		
`=>`	folding	`f=>k <x,y>`	≡	`f(x,f(y,k))`
`:-`	reduction	`f:-k <x,y,z,w>`	≡	`f(f(x,y),f(z,w))`
`<:`	recursive composition	`f<:g`	≡	`refer f+g`
`*^`	tree traversal	`~&dxPvV*^0`	≡	`~&dxPvVo`

Table 6.8: recursion combinators

6.8.1 Recursive composition

One operator from Table 6.8 that requires very little explanation is `<:`, for recursive composition. It has all four arities, no suffixes, and is fully dyadic. It is semantically equivalent to the composition operator, `+`, with the result wrapped in a `refer` combinator. That is, a function `f<:g` is equivalent to `refer f+g`. As noted previously, the `refer` combinator is used in recursively defined functions. An expression of the form `(refer f) x` evaluates to `f ~&J(f,x)`. See page 72 for more information.

6.8.2 Recursion over trees

The tree traversal operator, `*^`, is a generalization of the tree folding pseudo-pointer, `o`, introduced on page 70, that allows greater flexibility in the handling of empty subtrees, and accommodates arbitrary functional expressions as operands rather than literal pointer constants. It is useful for performing bottom-up calculations on trees.

The operator allows all arities and is prefix dyadic. The solo usage `*^` f is equivalent to the postfix usage f`*^`. A function of the form f`*^`k operates on a tree according to the following recurrence.

$$\begin{array}{rcl} \texttt{(}f\star\texttt{\textasciicircum}k\texttt{)}\ \ \texttt{\textasciitilde\&V()} &=& k \\ \texttt{(}f\star\texttt{\textasciicircum}k\texttt{)}\ \ d\texttt{\textasciicircum:<}v_0\ldots v_n\texttt{>} &=& f\,\texttt{(}d\texttt{\textasciicircum:<(}f\star\texttt{\textasciicircum}k\texttt{)}\ \ v_0\ldots\texttt{(}f\star\texttt{\textasciicircum}k\texttt{)}\ \ v_n\texttt{>)} \end{array}$$

A function f`*^` differs from f`*^`k by being undefined for the empty tree `~&V()` or any tree with an empty subtree.

The tree traversal operator allows a suffix consisting of any sequence of the characters `*` (asterisk), `.` (period), and =. Each of these characters specifies a transformation of the resulting function. The `*` makes it apply to every item of a list, the = composes it with a list flattening postprocessor, and the `.` makes it transform a list by deleting items that falsify it. When multiple characters occur in the same suffix, their effect is cumulative and the order matters.

6.8.3 Recursion over lists

The remaining two operators in Table 6.8 construct functions operating on lists according to patterns of recursion sometimes known as folding or reduction. A typical application for these operators is summing over a list of numbers.

Folding

The folding operator, `=>` takes a function operating on pairs of values and an optional constant as a vacuous case result to a function that operates on a list of values by nested applications of the function.

The operator can be used in any of four arities, with the infix form allowing a user defined vacuous case. It is prefix and solo dyadic, but the postfix form is without a vacuous case and consequently has a different semantics. There are currently no suffixes defined for it.

A function expressed as $f\texttt{=>}k$, which is equivalent to $(\texttt{=>}k)\ f$ and $(\texttt{=>})\ (f,k)$ by the dyadic properties, applies the following recurrence to a list.

$$
\begin{aligned}
(f\texttt{=>}k)\ \ \texttt{<>} &= k \\
(f\texttt{=>}k)\ \ h\texttt{:}t &= f(h,(f\texttt{=>}k)\ t)
\end{aligned}
$$

If f were addition and k were 0, this function would compute a cumulative sum. Cumulative products might conventionally have a vacuous case of 1. A function expressed by the postfix form $f\texttt{=>}$ is evaluated according to this recurrence.

$$
\begin{aligned}
(f\texttt{=>})\ \ \texttt{<>} &= \texttt{<>} \\
(f\texttt{=>})\ \ \texttt{<}h\texttt{>} &= h \\
(f\texttt{=>})\ \ h\texttt{:}t\texttt{:}u &= f(h,(f\texttt{=>})\ \ t\texttt{:}u)
\end{aligned}
$$

This form tends to have unexpected applications in *ad hoc* transformations of data, such as converting a list of length n to an n-tuple by `~&=>` (cf. Figures 5.1 and 5.2).

Reduction

The reduction operator, `:-`, performs a similar operation to folding, but the nesting of function applications follows a different pattern, and the vacuous case result doesn't enter into the calculation unnecessarily. The difference is illustrated by these two examples, which fold and reduce the operation of concatenation followed by parenthesizing with an empty vacuous case.

```
$ fun --m="-+'('--,--')',--+-=>'' ~&iNCS 'abcdefgh'" --c
'(a(b(c(d(e(f(g(h))))))))'
$ fun --m="-+'('--,--')',--+-:-'' ~&iNCS 'abcdefgh'" --c
'(((ab)(cd))((ef)(gh)))'
```

The original motivation for the reduction operator as opposed to folding was to avoid imposing unnecessary serialization on the computation. The current virtual machine implementation does not exploit this capability.

Algebraically the reduction operator has all four arities, no suffixes, and is fully dyadic (i.e., the vacuous case must always be specified). Semantically it may be regarded either as folding with an unspecified order of evaluation, limiting it to associative operations, or can have a formal specification consistent with above example, as documented for the

<table>
<tr><th></th><th>meaning</th><th>illustration</th><th></th><th></th></tr>
<tr><td><code>$^</code></td><td>maximizer</td><td><code>nleq$^ <1,2,3></code></td><td>≡</td><td><code>3</code></td></tr>
<tr><td><code>$-</code></td><td>minimizer</td><td><code>nleq$- <1,2,3></code></td><td>≡</td><td><code>1</code></td></tr>
<tr><td><code>-<</code></td><td>sort</td><td><code>nleq-< <2,1,3></code></td><td>≡</td><td><code><1,2,3></code></td></tr>
<tr><td><code>*~</code></td><td>filter</td><td><code>~=`x*~ 'axbxc'</code></td><td>≡</td><td><code>'abc'</code></td></tr>
<tr><td><code>~|</code></td><td>distributing filter</td><td><code>~=~| (`a,'bac')</code></td><td>≡</td><td><code>'bc'</code></td></tr>
<tr><td><code>|=</code></td><td>partition</td><td><code>==|= 'mississippi'</code></td><td>≡</td><td><code><'m','ssss','pp','iiii'></code></td></tr>
<tr><td><code>!=</code></td><td>bipartition</td><td><code>~=`x!= 'axbxc'</code></td><td>≡</td><td><code>('abc','xx')</code></td></tr>
<tr><td><code>*|</code></td><td>distributing bipartition</td><td><code>==*| (`a,'bac')</code></td><td>≡</td><td><code>('a','bc')</code></td></tr>
<tr><td><code>-~</code></td><td>forward bipartition</td><td><code>==`x-~ 'xax'</code></td><td>≡</td><td><code>('x','ax')</code></td></tr>
<tr><td><code>~-</code></td><td>backward bipartition</td><td><code>==`x~- 'xax'</code></td><td>≡</td><td><code>('xa','x')</code></td></tr>
</table>

Table 6.9: list combinators with predicate operands

`reduce` combinator in the `avram` reference manual.[7] A restricted form of this operation is provided by the `K21` pseudo-pointer explained on page 89.

6.9 List transformations induced by predicates

Some operators shown in Table 6.9 are designed to support frequently needed list calculations such as sorting, searching, and partitioning. A common feature of these operators is that they specify a function by a predicate or a boolean valued binary relation. Except as noted, all of these operators apply equally well to lists and sets.

6.9.1 Searching and sorting

Searching a list for an extreme value can be done by either of two operators, `$^` and `$-`, while sorting a list can be done by the `-<` operator. Searching is semantically equivalent to sorting followed by extracting the head of the sorted list, but is more efficient, requiring only linear time. Each of these operators requires a binary relational predicate and optionally a pointer or pseudo-pointer identifying a field on which to base the comparison.

A binary relational predicate p for these purposes is any function that takes a pair of values as an argument and returns a non-empty result if and only if the left value precedes the right according to some transitive relation. That is, $p(x, y)$ is true if and only if $x \sqsubseteq y$ for a relation $\sqsubseteq$. Examples of suitable relations are $\leq$ on floating point numbers as computed by `fleq` from the `flo` library, and alphabetic precedence on character strings as computed by `lleq` from the standard library, `std.avm`. The example `nleq` used in Table 6.9 is the partial order relation on natural numbers.

The pointer operand f can be any literal or symbolic expression evaluating to a pointer, including literals such as `&thl` or `&hthPX`, field identifiers such as `foobar`, or combinations of them such as `foobar.(&h:&tt)`. Pseudo-pointers are also acceptable, such as `&zl` or `foo.&iNC`.

[7]For a reduction combinator defined *ab initio* as a one-liner, see the file `com.fun` in the compiler source directory.

Semantics

The maximizing and minimizing functions cause an exception when applied to empty lists, but sorting an empty list is acceptable.

- The maximizing function `p$^f` applied to a list $<x_0 \ldots x_n>$ returns the item x_i for which `~f` x_i is the maximum with respect to the relation p.
- The minimizing function `p$-f` applied to a list $<x_0 \ldots x_n>$ returns the item x_i for which `~f` x_i is the minimum with respect to the relation p.
- The sorting function `p-<f` applied to a list $<x_0 \ldots x_n>$ returns a permutation of the list in which `~f` of each item precedes that of its successor with respect to the predicate p.

Algebraic properties

None of these operators is dyadic, but they can be used in all four arities and have similar algebraic properties

Postfix usage The postfix form of any of these operators, such as `p-<`, `p$-`, or `p$^`, is semantically equivalent to the infix form with a right operand of the identity pointer, `p-<&`, *etcetera*. That means the whole items of the argument list are compared to one another by p rather than a particular field f thereof.

Solo usage The solo usages `(-<) p`, `($^) p`, and `($-) p` are equivalent to the respective postfix usages `p-<`, `p$^`, and `p$-`. That is, they imply an identity pointer in place of the right operand and base the comparison on whole items of the list.

Prefix usage The prefix form of the sorting operator, `-<f` is equivalent to `lleq-<f`, where `lleq` is the lexical total order relation on character strings, and also the relation used by the compiler to represent sets as ordered lists.

The prefix forms of the maximizing and minimizing operators `$^f` and `$-f` are equivalent to `leql$^f` and `leql$-f` respectively, where `leql` is the relational predicate that tests whether one list is less or equal to another in length. The standard library defines `leql` as `~&alZ^!~&arPfabt2RB`.

Suffixes

Each of these operators allows a suffix, which can be any literal pointer or pseudo-pointer constant to be used as a postprocessor. That is, `p-<sf` with a pointer expression s is equivalent to `~&s+ p-<f`. Consequently, if the right operand f to a sorting or searching operator begins with an alphabetic character, it must be parenthesized to distinguish it from a suffix.

6.9.2 Filtering

The operation of filtering a list is that of transforming it to a sublist of itself wherein every item that falsifies a given predicate is deleted. Some operators previously introduced, such as composition and binary to unary combinators, can specify filtering functions by way of their suffixes, and filtering can also be done by the pseudo-pointers `F`, `K16`, and `K17`, but there are two operators intended specifically for filtering.

- The filter operator `*~` takes a predicate as an operand, and constructs a function that filters a list by deleting items that falsify the predicate (i.e., for which the predicate has an empty value).
- The distributing filter operator `~|` takes a binary relational predicate p as an operand (not necessarily transitive) and constructs a function that takes a pair $(a, \langle x_0 \dots x_n\rangle)$ to the sublist of the right argument containing only those x_i for which $p(a, x_i)$ is non-empty.

One way of thinking about these operators is that `*~` is used when the filtering criterion can be hard coded and `~|` is used when it's partly data dependent.

Usage

These operators can be used as follows.

- The `~|` operator is usable in any arity, and `*~` can be infix, postfix, or solo.
- In the prefix and infix usages, the right operand is a pointer expression.
- Both operators allow a pointer constant as a suffix, which serves as a postprocessor.
- The right operand, if any, must be parenthesized to distinguish it from a suffix if it begins with an alphabetic character.

Algebraic properties

Neither operator is dyadic, but the following algebraic properties hold, where p is a predicate and f is a pointer expression.

- The prefix usage of distributing bipartition implies a predicate of equality.

$$\mathtt{\sim|}\,f \;\equiv\; \mathtt{(==)\sim|}\,f$$

- The postfix usage of either operator is equivalent to the infix usage with an identity pointer as the right operand.

$$p\mathtt{*\sim} \;\equiv\; p\mathtt{*\sim\&}$$

- The postfix usage of either operator has an equivalent solo usage.

$$p\mathtt{*\sim} \;\equiv\; (\mathtt{*\sim})\,p$$

- The infix usage of either operator has an equivalent postfix usage.

$$p\mathtt{*\sim}f \;\equiv\; (p\mathtt{+\ \sim}f)\mathtt{*\sim}$$

Semantics

It is possible to supplement the informal descriptions above with rigorous definitions of these operators in various ways. The `*~` in postfix and solo forms without a suffix directly corresponds to the virtual machine's `filter` combinator, as documented in the `avram` reference manual. Alternatively, we may define

$$p\texttt{*\~{}}sf \;\equiv\; \texttt{\~{}\&}s\texttt{+ *= \&\&\~{}\&iNC } p\texttt{+ \~{}}f$$
$$p\texttt{\~{}|}sf \;\equiv\; \texttt{\~{}\&}s\texttt{+ \~{}\&rS+ } p\texttt{*\~{}}f\texttt{+ -*}$$

using operators defined elsewhere in this chapter, where p is a predicate, f is a pointer expression and s is a literal pointer or pseudo-pointer constant. Definitions for other arities are implied by the algebraic properties.

As indicated by these relationships, there is a minor point of difference between the usage of the pointer operand f with these operators and the sorting and searching operators described previously. In the present case, `~`f is applied to a pair of values, and its result is fed to p. In the previous case, `~`f is applied only to items of a list individually, and the pairs of its results are fed to p. The latter is more appropriate when p is a relational predicate, as with sorting and searching, whereas the present alternative is more general.

6.9.3 Bipartitioning

Bipartitioning is the operation of transforming a set S to a pair of subsets (L, R) such that $L \cap R$ is empty and $L \cup R = S$. It can also apply where S is a list, in which case the items of L and R preserve their order and multiplicity.

The bipartition operator `!=` shown in Table 6.9 takes a predicate p that is applicable to elements of a list or set S and constructs a function that bipartitions S into (L, R) such that p is true of all elements of L and false for all elements of R. This operator is documented further below, along with several related operators `*|`, `-~`, and `~-` also shown in Table 6.9. Pseudo-pointers with similar semantics are documented in Section 2.5.2.

Bipartition

The `!=` operator can be used in any of prefix, infix, postfix, and solo arities. The left operand, if any, is a predicate and the right operand, if any, is a pointer or pseudo-pointer expression. The operator may also have a literal pointer constant as a suffix. If there is a right operand beginning with an alphabetic character, it must be parenthesized to distinguish it from a suffix.

Algebraic properties The following algebraic properties hold, where p is a predicate and f is a pointer expression.

- The postfix usage implies the identity as a pointer operand.

$$p\texttt{!=} \;\equiv\; p\texttt{!=\&}$$

- The prefix usage implies the identity function as a predicate.

$$\texttt{!=}f \;\equiv\; \texttt{\~{}\&!=}f$$

- The infix usage is defined by the solo usage.

$$p\texttt{!=}f \;\equiv\; (\texttt{!=})\ p\texttt{+}\ \ \texttt{\~{}}f$$

Semantics It is straightforward to give a formal semantics for the postfix arity (and the others by implication) in terms of the $\texttt{\~{}\&j}$ pseudo-pointer for set difference and the filter combinator.

$$(p\texttt{!=})\ x = ((\texttt{!=})\ p)\ x = \texttt{(}(p\star\texttt{\~{}})\ x\texttt{,\~{}\&j/}x\ \ (p\star\texttt{\~{}})\ x\texttt{)}$$

The optional suffix serves as a postprocessor in any arity. For a pointer constant s, any function of the form $p\texttt{!=}sf$, $\texttt{!=}sf$, $p\texttt{!=}s$, or $\texttt{!=}s$. is equivalent to $\texttt{\~{}\&}s\texttt{+}\ \ g$, where g is given by $p\texttt{!=}f$, $\texttt{!=}f$, $p\texttt{!=}$, or $\texttt{!=}$ respectively.

Distributing bipartition

The distributing bipartition operator $\star\texttt{|}$ is used to bipartition a list according to a binary relation. A function $p\star\texttt{|}f$ takes pair of $\texttt{(}x\texttt{,<}y_0\dots y_n\texttt{>)}$ as an argument, and it returns a pair of lists $\texttt{(<}y_i\dots\texttt{>,<}y_j\dots\texttt{>)}$ collectively containing all of the items y_0 through y_n. For all y_i in the left side of the result, $p\ \ \texttt{\~{}}f\ \ (x, y_i)$ has a non-empty value (using the same x in every case). For all y_j in the right side, $p\ \ \texttt{\~{}}f\ \ (x, y_j)$ has an empty value.

This operator has the same algebraic properties and arities as the bipartition operator discussed above, and makes similar use of an optional pointer expression as a suffix. Its semantics is given by

$$p\star\texttt{|}sf \;\equiv\; \texttt{\~{}\&}s\texttt{+}\ \ \texttt{\~{}\&brS+}\ \ p\texttt{!=}f\texttt{+}\ \ \texttt{-}\star$$

where the suffix s is a literal pointer constant and f is any pointer expression, possibly parenthesized.

Ordered bipartition

The two operators, $\texttt{-\~{}}$ and $\texttt{\~{}-}$, are used for bipartitioning a list S based on a predicate p into a pair of lists (L, R) such that S is the concatenation of L and R.

- A function $p\texttt{-\~{}}$ applied to S will construct (L, R) with L as the maximal prefix of S whose items all satisfy p.
- A function $p\texttt{\~{}-}$ will make R the maximal suffix whose items all satisfy p.

In operational terms, $p\texttt{-\~{}}$ scans forward through a list from the head and stops at the first item for which p is false, whereas $p\texttt{\~{}-}$ scans backwards from the end. The results may or may not coincide with each other or with $p\texttt{!=}$ depending on repetitions in S and the semantics of p.

These operators allow solo usages, with $(\texttt{-\~{}})\ p$ equivalent to $p\texttt{-\~{}}$, and $(\texttt{\~{}-})\ p$ equivalent to $p\texttt{\~{}-}$, and they each allow a pointer suffix to specify a postprocessor.

6.9.4 Partitioning

The partition operator, `|=`, shown in Table 6.9 can be used to identify equivalence classes of items in a list or a set according to any given equivalence relation, or by the transitive closure of any given relation. This operator is very expressive, for example by allowing a function locating clusters or connected components in a graph to be expressed simply in terms of a suitable distance metric or adjacency relation.

Usage

The partition operator can be used in prefix, postfix, infix, and solo arities. In the prefix and infix arities, the right operand is a pointer expression. In the postfix and infix arities, the left operand is a binary relational predicate. There may also be a a suffix in any arity consisting of a sequence of the characters `=`, `*`, or a literal pointer constant. The right operand, if any, must be parenthesized to distinguish it from a suffix if it begins with an alphabetic character.

Algebraic properties

The operator is not dyadic, but has these properties, which also hold when it has a suffix.

- The prefix usage implies a relational predicate of equality by default.

$$\texttt{|=}f \equiv \texttt{(==)}\,\texttt{|=}f$$

- The postfix usage implies the identity pointer by default.

$$p\,\texttt{|=} \equiv p\,\texttt{|=\&}$$

- The infix usage can be defined by the solo usage.

$$p\,\texttt{|=}f \equiv (\texttt{|=})\,(p\texttt{+ \~{}\&b.}f)$$

- The postfix usage $p\,\texttt{|=}$ is equivalent to the solo usage $(\texttt{|=})\,p$ because $p\texttt{+ \~{}\&b.\&}$ is equivalent to p when p is a binary predicate.

Semantics

Intuitively, the relational predicate p in a function $p\,\texttt{|=}$ is true of any pair of values that belong together in the same partition. and the pointer f identifies a field within each list item to be compared by p.

The relation should be an equivalence relation, which by definition is reflexive, transitive and symmetric, but if the latter two properties are lacking, the operator can be invoked in such a way as to compensate. An example of an equivalence relation is that of two words being equivalent if they begin with the same letter. Usually any rule associating two things that share a common property induces an equivalence relation.

<table>
<tr><th></th><th>meaning</th><th>illustration</th><th></th><th></th></tr>
<tr><td><code>*</code></td><td>map</td><td><code>f* <a,b></code></td><td>≡</td><td><code><f a,f b></code></td></tr>
<tr><td><code>~*</code></td><td>map to both</td><td><code>f~* (x,y)</code></td><td>≡</td><td><code>(f* x,f* y)</code></td></tr>
<tr><td><code>*=</code></td><td>flattening map</td><td><code>f*= <a,b></code></td><td>≡</td><td><code>~&L <f a,f b></code></td></tr>
<tr><td><code>|\</code></td><td>triangle combinator</td><td><code>f|\ <a,b,c></code></td><td>≡</td><td><code><a,f b,f f c></code></td></tr>
<tr><td><code>^</code></td><td>coupling</td><td><code>^(f,g) x</code></td><td>≡</td><td><code>(f x,g x)</code></td></tr>
<tr><td><code>~~</code></td><td>apply to both</td><td><code>f~~ (x,y)</code></td><td>≡</td><td><code>(f x,f y)</code></td></tr>
<tr><td><code>^~</code></td><td>couple and apply to both</td><td><code>f^~(g,h) x</code></td><td>≡</td><td><code>(f g x,f h x)</code></td></tr>
<tr><td><code>^*</code></td><td>mapped coupling</td><td><code>f^*(g,h)</code></td><td>≡</td><td><code>f*+ ^(g,h)</code></td></tr>
<tr><td><code>^|</code></td><td>apply one to each</td><td><code>^|(f,g) (x,y)</code></td><td>≡</td><td><code>(f x,g y)</code></td></tr>
<tr><td><code>$</code></td><td>record lifter</td><td><code>rec$[a: f,b: g]</code></td><td>≡</td><td><code>^(f,g)</code></td></tr>
</table>

Table 6.10: concurrent forms

This explanation can be made more rigorous in the following way. For the postfix arity, the `|=` operator satisfies this recurrence up to a re-ordering.

$$\begin{aligned}(p\,|\!=)\ \mathtt{<>} &= \mathtt{<>}\\ (p\,|\!=)\ h\mathtt{:}t &= \mathtt{:\hat{}(:/}h\mathtt{+\ \tilde{}\&lL,\tilde{}\&r)}\ p\tilde{}\,|\star|/h\ (p\,|\!=)\ t\end{aligned}$$

The semantics for other arities follows from the algebraic properties above. The coupling operator, `^`, is introduced subsequently in this chapter. The subexpression p`~|*|/`h is parsed as `((`p`~|)*|)/`h to use a distributing filter within a distributing bipartition as the left operand of a binary to unary operator.

- If there is a suffix that includes the = character (e.g. if the operator is of the form `|==`), the symmetric closure of the predicate p is implied, and the above recurrence holds with `-!`p`,`p`+~&rlX!-~|` in place of p`~|`.
- A function of the form p`|=`s, p`|==`s, p`|=*`s, or p`|=*=`s, where s is a literal pointer or pseudo-pointer constant, is semantically equivalent to a function `~&`s`+` g, where g is of the form p`|=`, p`|==`, p`|=*`, or p`|=*=` respectively.
- If there is *not* a suffix containing the `*`, the above recurrence accurately describes the semantics only if p is transitive (i.e., if $p(x, y)$ and $p(y, z)$ implies $p(x, z)$). If there is a suffix containing `*`, the recurrence holds regardless of transitivity.

A more efficient algorithm is used for partitioning when the relation p is transitive, but unspecified results are obtained if this algorithm is used when p is not transitive. If p is not transitive, it is the user's responsibility to specify the `*` in a suffix. An example of a relation that is not transitive is intersection between sets.

6.10 Concurrent forms

Whatever the merits of functional programming for concurrent applications, the operators in Table 6.10 are variations on the theme of computations with obvious parallel evaluation strategies. Although the virtual machine makes no use of parallelism in its present implementation, these operators are convenient as programming constructs for their own sake. They fall broadly into the classifications of mapping operators and coupling operators, which are considered separately in this section.

6.10.1 Mapping operators

The first four operators in Table 6.10 involve making a list of outputs from a function by applying the function to every item of an input list. They can be used either in solo arity, or as a postfix operator with a function as an operand, and they share the algebraic property $f\star \equiv (\star)\ f$. They also have suffixes usable in various ways.

Map The simplest and most frequently used mapping operator, $\star$, satisfies this recurrence when used without a suffix.

$$\begin{aligned} (f\star)\ \texttt{<>} &= \texttt{<>} \\ (f\star)\ h\texttt{:}t &= (f\ h)\texttt{:}((f\star)\ t) \end{aligned}$$

That is, the map of f applies f to every item of its input list and returns a list of the results. Mapping can also be used on sets but the result should be regarded as a list unless uniqueness and lexical ordering of the items in the result are maintained, which are necessary invariants for the set representation.

The $\star$ operator allows a literal pointer constant as a suffix, and the suffix serves as a preprocessor to the mapping function (not a postprocessor as it does for most other operators allowing pointer suffixes). For a literal pointer s, the relationship is

$$f{\star}s \equiv f{\star}\texttt{+ \~{}\&}s$$

Pseudo-pointers as suffixes for the map operator can be very expressive. For example, a matrix multiplication function can be defined in one line as

```
mmult = (plus:-0.+ times*p)*rlD*rK7lD
```

using either `plus` and `times` from the `flo` library with floating point 0, or whatever equivalents are appropriate for matrices over some other field.

Map to both The $\texttt{\~{}}\star$ operator works like the $\star$ operator except that it constructs a function that applies to a pair of lists rather than a single list. The exact relationship is

$$(f\star\texttt{\~{}})\ (x,y) \equiv ((f\star)\ x, (f\star)\ y)$$

where f is a function and x and y are lists. This operator also allows a pointer suffix, that serves as a preprocessor That is,

$$f\star\texttt{\~{}}s \equiv \texttt{\~{}\&}s\texttt{;}\ f\star\texttt{\~{}}$$

where s is a literal pointer constant.

Flattening map The $\star$= operator behaves like the $\star$ with a list flattening postprocessor. The function f in an expression $f\star$= should return a list. After making a list of the results, which will be a list of lists, the flattening map operation forms their cumulative concatenation. Formally, the relationship is

$$f\star\texttt{=} \;\equiv\; \texttt{\~{}\&L+}\;\; f\star$$

in terms of the list flattening pseudo-pointer `~&L` explained on page 65, which could also be defined as `--:-<>` with operators introduced in this chapter.

The flattening map operator allows arbitrarily many more $\star$ and = characters to be appended as suffixes.

- Each $\star$ character in a suffix indicates a nested map. That is, $f\star\texttt{=}\star$ is equivalent to $(f\star\texttt{=})\star$, where the latter $\star$ is parsed as the map operator, $f\star\texttt{=}\star\star$ is equivalent to $((f\star\texttt{=})\star)\star$, and so on.
- Each = character in a suffix indicates another iteration of flattening. Hence $f\star\texttt{==}$ is equivalent to `~&L+` $f\star$=, and $f\star\texttt{===}$ is equivalent to `~&L+ ~&L+` $f\star$=, and so on.
- Combinations of these characters within the same suffix are allowed but the order matters. $f\star\texttt{=}\star\texttt{=}$ is equivalent to `~&L+` $(f\star\texttt{=})\star$, which is also equivalent to a pair of nested flattening maps $(f\star\texttt{=})\star\texttt{=}$, but $f\star\texttt{==}\star$ is equivalent to (`~&L+` $f\star$=)$\star$.

A pointer expression may also appear in a suffix, and it will act as a preprocessor similarly to a pointer suffix for the map operator.

Triangulation An operator that is less frequently used but elegant when appropriate is the `|\` operator for triangulation. This operator should not be confused with `/|` or `\|`, the binary to unary combinators with a suffix of `|`, although the meanings are related (page 202). See also the `K9` pseudo-pointer on page 87.

The intuitive description of the triangle combinator is that it takes a function f as an operand and constructs a function that transforms a list as follows.

$$(f\,|\backslash)\;\texttt{<}x_0\texttt{,}\,x_1\texttt{,}\,x_2\texttt{,}\quad\ldots x_n\texttt{>} = \texttt{<}x_0\texttt{,}\,f(x_1)\texttt{,}\,f(f(x_2))\texttt{,}\quad\ldots\;\underbrace{f(\ldots f(}_{n\text{ times}}x_n)\ldots)\texttt{>}$$

That is, the function f is applied i times to the i-th item of the list. A more formal description would be that it satisfies the following recurrence.

$$\begin{aligned}(f\,|\backslash)\;\texttt{<>} &= \texttt{<>}\\ (f\,|\backslash)\;h\texttt{:}t &= h\texttt{:}((f\,|\backslash)\;\;(f\star)\;\;t)\end{aligned}$$

The triangle combinator also allows a literal pointer or pseudo-pointer constant s as a suffix, which serves as a postprocessor.

$$f\,|\backslash s \;\equiv\; \texttt{\~{}\&}s\texttt{+}\;\; f\,|\backslash$$

6.10.2 Coupling operators

Whereas the mapping operators are concerned with applying the same function to multiple arguments, most of the remaining operators in Table 6.10 involve concurrently applying multiple functions to the same argument.

Apply to both

The `~~` operator allows postfix and solo arities with no suffixes. In the postfix arity, its operand is a function, and the solo arity satisfies $(\tilde{}\,\tilde{})\ f \equiv f\tilde{}\,\tilde{}$.

This operator corresponds to what is called the `fan` combinator in the `avram` reference manual. Given a function f, it constructs a function that applies to a pair of values and returns a pair of values. Each side of the output pair is computed by applying f to the corresponding side of the input pair.

$$(f\tilde{}\,\tilde{})\ (x, y)\ \equiv\ (f\ x, f\ y)$$

Normally a function of the form $f\tilde{}\,\tilde{}$ will raise an exception with a diagnostic message of "`invalid deconstruction`" when applied to an empty argument, but if the function f is of the form `~&`p and p is a pointer, certain code optimizations might apply.

```
$ fun --main="~&~~" --decompile
main = field &
$ fun --m="~&rlX~~" --d
main = field((((0,&),(&,0)),0),(0,((0,&),(&,0))))
```

The optimization in the first example is a refinement rather than an equivalent semantics, whereby the function will map an empty input to an empty output rather than raising an exception. The optimization in the second example uses a single pointer instead of the `fan` combinator.

This operator also allows a pointer suffix, that serves as a preprocessor That is,

$$f\tilde{}\,\tilde{}s\ \equiv\ \tilde{}\&s;\ \ f\tilde{}\,\tilde{}$$

where s is a literal pointer constant.

Couple

The most frequently used coupling combinator is `^`, which allows infix, postfix, and solo arities, and a pointer suffix as a postprocessor.

- In the solo arity, `^` is a function that takes a pair of functions as an argument and returns a function as a result.
- In the infix arity, the `^` operator takes a function as its left operand and a pair of functions as its right operand, with the algebraic property $f\,\hat{}\,(g, h) \equiv f\!+\ (\hat{}\,)(g, h)$.
- The operator is postfix dyadic, so the postfix usage is implied by the infix.

The semantics for the solo arity, which implies the other two, is given by

$$((\hat{}\ )\ \ (f,g))\ \ x\ \equiv\ (f\ x, g\ x)$$

where f and g are functions. That is, a function $\hat{}\,(f,g)$ returns a pair whose left side is computed by applying f to the argument, and whose right side is computed by applying g to the argument. This operation corresponds to the virtual machine's `couple` combinator.

The interpretation of a pointer suffix s varies depending on the arity.

- In the solo arity, the suffix acts as a postprocessor to the function that is constructed.

$$\hat{}\,s(f,g)\ \equiv\ \tilde{}\,\&s+\ \ \hat{}\,(f,g)$$

- In the infix arity, the suffix is composed between the left operand and the function constructed from the right operands.

$$f\,\hat{}\,s(f,g)\ \equiv\ f+\ \ \tilde{}\,\&s+\ \ \hat{}\,(f,g)$$

- Suffixes in the postfix arity function consistently with the infix arity.

$$(h\,\hat{}\,s)\ (f,g)\ \equiv\ h\,\hat{}\,s(f,g)$$

Compound coupling

The two operators $\hat{}\,\tilde{}$ and $\hat{}\,\star$ perform a combination of the $\hat{}$ with the $\tilde{}\,\tilde{}$ and $\star$ operations, respectively. They allow infix, postfix, and solo arities, and have these algebraic properties.

- The infix usage of $\hat{}\,\tilde{}$ causes the left operand to be applied to both results returned by the function constructed from the right operand.

$$f\,\hat{}\,\tilde{}\,(g,h)\ \equiv\ f\,\tilde{}\,\tilde{}+\ \ \hat{}\,(g,h)$$

- The infix usage of $\hat{}\,\star$ has the analogous property, but is not well typed unless a pseudo-pointer suffix transforms the intermediate result to a list (see below).

$$f\,\hat{}\,\star(g,h)\ \equiv\ f\star+\ \ \hat{}\,(g,h)$$

- Both operators are postfix dyadic.

$$\begin{aligned}(f\,\hat{}\,\tilde{}\,)\ (g,h)\ &\equiv\ f\,\hat{}\,\tilde{}\,(g,h)\\ (f\,\hat{}\,\star)\ (g,h)\ &\equiv\ f\,\hat{}\,\star(g,h)\end{aligned}$$

- The solo usage takes a function as an argument and returns a function that takes a pair of functions as an argument.

$$\begin{aligned}(\hat{}\,\tilde{}\ f)\ (g,h)\ &\equiv\ f\,\hat{}\,\tilde{}\,(g,h)\\ (\hat{}\,\star\ f)\ (g,h)\ &\equiv\ f\,\hat{}\,\star(g,h)\end{aligned}$$

If a pointer constant s is used as a suffix, it is composed between the `fan` or map of the left operand and the functions constructed from the right operand.

$$\begin{array}{rcl} f\texttt{\^{}\~{}}s(g,h) & \equiv & f\texttt{\~{}\~{}+ \~{}\&}s\texttt{+ \^{}}(g,h) \\ f\texttt{\^{}}\star s(g,h) & \equiv & f\star\texttt{\^{}+ \~{}\&}s\texttt{+ \^{}}(g,h) \end{array}$$

The semantics of pointer suffixes in the other arities of these operators is analogous to those of the `^` operator.

One to each

A further variation on the couple operator is `^|`. The semantics in the infix arity with a pointer suffix s is

$$(f\texttt{\^{}|}\,s(g,h))\;(x,y)\;\equiv\;f\;\texttt{\~{}\&}s\;\;(g\;x,h\;y)$$

where f, g, and h are functions. The solo arity satisfies

$$((\texttt{\^{}|}\,s)\;(g,h))\;(x,y)\equiv\;\texttt{\~{}\&}s\;\;(g\;x,h\;y)$$

and the operator is postfix dyadic.

If a function of the form $f\texttt{\^{}|}\,s(g,h)$ is applied to an empty value instead of a pair (x,y), an exception will be raised with "`invalid deconstruction`" reported as a diagnostic. Otherwise, one function is applied to each side of the pair, as the above equivalence indicates.

In addition to a pointer suffix s, this operator may be used with any combination of suffixes $\star$, `=`, and `~`. The simplest way of understanding and remembering their effects is by these identities,

$$\begin{array}{rcl} f\texttt{\^{}|}\star s(g,h) & \equiv & (f\star)\texttt{\^{}|}\,s(g,h) \\ f\texttt{\^{}|\~{}}s(g,h) & \equiv & (f\texttt{\~{}\~{}})\texttt{\^{}|}\,s(g,h) \\ f\texttt{\^{}|}\star\texttt{=}s(g,h) & \equiv & (f\star\texttt{=})\texttt{\^{}|}\,s(g,h) \end{array}$$

which is to say that they can be envisioned as making the left function mapped, fanned, or flat mapped. These suffixes may also be used in the solo form, wherein they act on the implied identity function instead of a left operand. The flattening suffix, `=`, can be used by itself, and will have the effect of composing the list flattening function `~&L` with the left operand. Arbitrarily long sequences of these suffixes are also allowed, and are applied in order, as in this example.

$$f\texttt{\^{}|}\star\texttt{\~{}=}\star s(g,h)\equiv(\star\;\texttt{\~{}\&L+}\;\;\texttt{\~{}\~{}}\star\;f)\texttt{\^{}|}\,s(g,h)$$

Record lifting

For records to be useful as abstract data types, the capability to manipulate them without recourse to the concrete representation is essential. This requirement is partly filled by the means documented in Section 4.2 for declarations and deconstruction of record types and instances, but further support is needed for their dynamic creation and transformation.

The $ operator is used to express functions returning records in an abstract style, while preserving any invariants stipulated in the record's declaration. It allows postfix and solo arities, with the property $f\texttt{\$} \equiv (\texttt{\$})\ f$. Nested $ operators in expressions such as $f\texttt{\$\$}$ and $f\texttt{\$\$\$}$ are meaningful as higher order functions. The operand f can be any function, but only functions defined by record declarations are likely to be useful (i.e., defined as the initializing function denoted by the record mnemonic). The $ operator also allows a pointer constant as a suffix, which is used in an unusual way explained presently.

Usage A function of the form $f\texttt{\$}$ with a record mnemonic f is analogous to a function $g\texttt{\^{}}$ for a function g operating on a pair of values. Whereas the latter is meaningful when applied to a pair of functions (as explained in connection with the ^ operator), the former applies to a record of functions. Hence, the typical usage is in an expression of the form

$$
\begin{array}{l}
\langle\textit{record mnemonic}\rangle\texttt{\$[}\\
\qquad\qquad\langle\textit{field identifier}\rangle\texttt{:}\quad\langle\textit{function}\rangle\texttt{,}\\
\qquad\qquad\qquad\qquad\vdots\\
\qquad\qquad\langle\textit{field identifier}\rangle\texttt{:}\quad\langle\textit{function}\rangle\texttt{]}
\end{array}
$$

which is parsed as $(\langle\textit{record mnemonic}\rangle\texttt{\$)[...]}$. The record mnemonic and field identifiers should match those of a record type previously declared with the :: operator, as explained in Section 4.2.

- The fields in a record valued function can be specified in any order or omitted, but at least one must be included.
- The effect of repeating a field in the same expression is unspecified, but in the current implementation one or another will take precedence.
- The technique of associating a tuple of values with a tuple of fields is *not* valid for record valued functions, even though it ordinarily can be used to express record instances. For example, the subexpression `[a: fa,b: fb]` should not be abbreviated to `[(a,b): (fa,fb)]` in a record valued function.

Semantics The $ operator can be understood by this equivalence.

$$((f\texttt{\$})\texttt{[}a_0\texttt{:}\ g_0\texttt{,}\ \ \dots\ a_n\texttt{:}\ g_n\texttt{]})\ x\ \equiv\ f\texttt{[}a_0\texttt{:}\ g_0(x)\texttt{,}\ \ \dots\ a_n\texttt{:}\ g_n(x)\texttt{]}$$

That is, $(f\texttt{\$})\texttt{[}a_0\texttt{:}\ g_0\texttt{,}\ \ \dots\ a_n\texttt{:}\ g_n\texttt{]}$ represents a function that can be applied to an argument x to return a record of the type indicated by f. To compute this function, each g_i is applied to the argument, and its result is stored in the field with address a_i in the manner portrayed in Figure 5.3 (page 189). The record of function results is then initialized by the record initializing function f. At this stage, any user defined verification or initialization specified in the record declaration is automatically performed, even if it overrules the function results.

Nested use of the operator denotes a higher order function.

$$\begin{array}{rcl} ((f\texttt{\$\$})\texttt{[}a_0\texttt{:}\ g_0\texttt{,}\ \ldots\ a_n\texttt{:}\ g_n\texttt{]})\ x & \equiv & (f\texttt{\$})\texttt{[}a_0\texttt{:}\ g_0(x)\texttt{,}\ \ldots\ a_n\texttt{:}\ g_n(x)\texttt{]} \\ ((f\texttt{\$\$\$})\texttt{[}a_0\texttt{:}\ g_0\texttt{,}\ \ldots\ a_n\texttt{:}\ g_n\texttt{]})\ x & \equiv & (f\texttt{\$\$})\texttt{[}a_0\texttt{:}\ g_0(x)\texttt{,}\ \ldots\ a_n\texttt{:}\ g_n(x)\texttt{]} \\ & \vdots & \end{array}$$

Although the semantics in higher orders is formally straightforward, lambda abstraction may be a more readable alternative in practice (page 207).

Suffixes Not every field defined when the record is declared has to be specified in a record valued function. This feature reduces clutter and allows easier code maintenance if more fields are added to a record in the course of an upgrade.[8] The handling of omitted fields depends on the optional pointer suffix to the `$` operator.

With no suffix, the default behavior of the `$` is to assign an empty value to an omitted field, but for a typed or smart record, the empty fields are automatically initialized by the record initializing function f.

If there is a pointer or pseudo-pointer suffix s appended to the `$` operator, then any omitted field a_i is assigned a value of $\texttt{\~{}}s\texttt{.}a_i\ \ x$, where x is the argument to the function. Intuitively that means that the unspecified fields in a result can be copied or inherited automatically from a record in the argument. This value may still be subject to change by the record initializing function.

By way of an example, a function taking a record of type `_foo` to a modified record of the same type with most of the fields other than `bar` unchanged could be expressed as `foo$i[bar: g]`. This function is almost equivalent to `bar:=`g using the assignment operator (page 203) except that it provides for the record to be reinitialized after the change. Other common usages are `$l` and `$r`, for functions that take a pair of a record and something else to a new record by copying mostly from the input record.

6.11 Pattern matching

A set of operators relevant to the general theme of pattern matching or transformation is shown in Table 6.11. They are classified in this section as random variate generators, type expression constructors, finite maps, and string handling operators.

6.11.1 Random variate generators

An operator in a class by itself is `%~`, which is useful for constructing programs with non-deterministic outputs. It can be used in postfix or solo arities, and has the property n`%~` $\equiv$ (`%~`) n. Its operand n is either a natural or a floating point number.

[8]If the declaration and use of a record are in separate modules, both may require recompilation even if no source level changes are made to the latter.

	meaning	illustration		
%~	bernoulli variable	50%~ x	≡	& or 0
%	literal type expressions	(%s,%t)%dlwrX	≡	%stX
%-	symbolic type expressions	%-u x	≡	x%u
-$	unzipped finite map	<a,b>-$<x,y> a	≡	x
-:	defaultable finite map	<a: x,b: y>-:d c	≡	d
=:	address map	<a: x,b: y>=: b	≡	y
%=	string replacement	'b'%='d' 'abc'	≡	'adc'
=]	startswith combinator	=]'ab' 'abc'	≡	true
[=	prefix combinator	[='abc' 'ab'	≡	true

Table 6.11: Pattern matching

Semantics

A program of the form n%~ can be used in place of a function but does not have a functional semantics. Rather, it ignores its argument and returns a boolean value, either 0 or &. The value it returns is obtained by simulating a draw from a random distribution. The operand n allows a distribution to be specified.

- If n is a floating point number, it should be between 0 and 1. Then n%~ will return a true value with probability n.
- If n is a natural number, it should range from 0 to 100, and n%~ will return a true value with probability $n/100$.
- A default probability of 0.5 is inferred for the usage 0%~.

The above probability should be understood as that of the simulated distribution. The results are actually obtained deterministically by the Mersenne Twister algorithm for random number generation provided by the virtual machine. In operational terms, if n%~ is applied to members of a population (i.e., items of a list), the percentage of true values returned will approach n as the number of applications increases.

Applications

This operator can be used for generating pseudo-random data of general types and statistical properties by using it in programs of the form n%~?(f,g), where f and g can be functions returning any type and can involve further uses of %~. However, a better organized approach for serious simulation work might involve the combinators arc and stochasm defined in the standard library. A more convenient method when the distribution parameters aren't critical is to use type instance generators (page 170).

Because n%~ is not a function, certain code optimizations based on the assumption of referential transparency are not applicable to it. The code optimization features of the compiler handle it properly without any user intervention required. However, developers of applications involving automated program transformation may need to be aware of it. See page 82 for a related discussion.

6.11.2 Type expression constructors

Two operators concerned with type expressions are topical for this section because type instance recognizers are an effective pattern recognition mechanism. Type expressions are a significant topic in themselves, being thoroughly documented in Chapters 3 and 4, but the operators `%-` and `%` are included here for completeness and because they have some previously unexplained features.

The `%` operator

The type operator `%` allows postfix and solo arities, with different meanings depending mainly on the suffix.

- If there is a suffix containing alphabetic characters, the operator represents a type expression or type induced function in either arity as documented in Chapters 3 and 4.
- If there is a suffix containing only numeric characters, then the operator represents an exception handler in the solo arity but is undefined in the postfix arity.
- If there is no suffix, it represents an exception generator in either arity, and has the property $f\texttt{\%} \equiv (\texttt{\%})\; f$.

The latter two alternatives require further explanation.

Exception handlers An expression of the form $\texttt{\%}n$, where n is a sequence of digits, is a higher order function meant to be applied to a function f. It will return a function g that behaves identically to f unless g is applied to an argument that would cause f to raise an exception. In that case, g will also raise an exception, but the content of the diagnostic message will differ from that which would be reported by f, in that the number n will be appended to it. A simple illustration is given by the following examples.

```
$ fun --m="~&h <>" --c
fun:command-line: invalid deconstruction
$ fun --m="(%52 ~&h) <>" --c
fun:command-line: invalid deconstruction
52
$ fun --m="~&h <'x'>" --c
'x'
$ fun --m="(%52 ~&h) <'x'>" --c
'x'
```

This usage of the operator is intended mainly for debugging applications that are terminating ungracefully, by helping to locate the problem. See Section 4.1.2 and particularly page 143 for background and motivation about exception handling.

Exception generators Although exceptions are usually associated with ungraceful termination, there could also be reasons for raising them deliberately in production code. The default case in a `-?...?-` cumulative conditional expression wherein the other cases are thought to be exhaustive is one example (page 191). Failure of an assertion is another.

An expression of the form `%` f or f`%`, where f is a function, represents a function that unconditionally raises an exception. The function f is applied to the argument, execution is either immediately terminated or dropped into an enclosing exception handler, and the result from f is reported in a diagnostic message.

Because diagnostic messages are written to the standard error console by the virtual machine, they should normally be lists of character strings (type `%sL`).

- If the function f returns something other than a list of character strings and the exception is raised during compilation, the compiler will substitute a diagnostic message of "`undiagnosed error`".
- If a badly typed diagnostic is reported in a free standing executable application, the virtual machine may report a diagnostic of "`invalid text format`" or attempt to display unprintable characters.
- Users who think it's worth the effort can throw diagnostics of arbitrary types and catch them using the virtual machine's `guard` combinator, provided the latter converts them to lists of character strings. This combinator is documented in the `avram` reference manual.

A frequently used idiom is an exception generator made from a function f returning a constant list of a single character string, as in `<'game over'>!%`. A more helpful alternative if possible is an exception handler that gives some indication of the input that caused the exception, such as `% :/'bad input was'+ %xP`, preferably with a more specific printing function than `%xP`.

Confusing effects can occur if the function f in an expression f`%` raises an exception itself either because of a programming error or because of a nested `%` operator. The reported diagnostic will then refer to the exception generator itself rather than the program containing it. Moreover, interaction between the exception generator and exception handlers or `guard` combinators will be affected because exceptions form a hierarchy of segregated levels. See the `avram` reference manual for more information.

The `%-` operator

This operator is unusual insofar as it allows only a solo arity, but may have a literal type expression as a suffix. It has the property

$$\texttt{\%-}t\;x \;\equiv\; x\texttt{\%}t$$

where t is a literal type expression constant or type induced function. It exists to provide a convenient means for general purpose functions to construct type expressions. For example, a user preferring a more verbose programming style might define

```
list_of = %-L
```

Listing 6.1 decompilation of optimal code generated by `<0,1,2,3,4,5,6,7>-$'01234567'`

```
digitize = # takes a number 0..7 to the corresponding digit

conditional(
   field &,
   conditional(
      field(&,0),
      conditional(
         field(0,&),
         conditional(
            field(0,(&,0)),
            conditional(field(0,(0,&)),constant `7,constant `3),
            constant `5),
         constant `1),
      conditional(
         field(0,(&,0)),
         conditional(field(0,(0,&)),constant `6,constant `2),
         constant `4)),
   constant `0)
```

and thereafter write `list_of(my_type)` instead of `my_type%L`. A more practical example is the `enum` function, which the standard library defines as

```
enum = ~&ddvDlrdPErvPrNCQSL2Vo+ %-U:-0+ %-u*
```

taking any non-empty set to an enumerated type thereof. The pseudo-pointer postprocessor is a low level optimization to the type expression's concrete representation, and not presently relevant. See page 173 for motivation.

6.11.3 Reification

A finite map is a function whose inputs are expected only to be members of a fixed finite set, usually something small enough to enumerate exhaustively like a set of mnemonics or numerical instruction codes. In some applications, a finite map turns out to be a "hot spot" that can improve performance if optimized.

There are three operators provided in support of finite maps. They generate code that is optimal in the sense of requiring minimally many interrogations on an amortized basis.[9] This effect is achieved by detecting differences between the concrete representations of the possible input values without regard for their types.

For example, the quickest function to convert natural numbers in the range `0` through `7` to the corresponding characters `` `0 `` through `` `7 `` would be the the one shown in Listing 6.1. In the worst case, five conditionals testing individual bits of the argument are evaluated, but in the best case, only one.[10] In any case, it would be irritating to develop or maintain

[9] I.e., the quick ones make up for the slow ones, but they're all pretty quick.

[10] Recall from page 115 that natural numbers are represented as arbitrary length lists of booleans lsb first, so both the length and the content must be established.

this code by hand, which is the motivation for reification operators.

Algebraic properties

The three reification operators are `-:`, `-$`, and `=:`, for zipped finite maps, unzipped finite maps, and address maps.

- The `-$` operator can be used in any arity and is fully dyadic.
- The `-:` operator can also be used in any arity. It is prefix and postfix dyadic, but has the solo semantics described below.
- The `=:` operator can be used in postfix or solo arities, and satisfies $m\texttt{=:}\ \equiv\ (\texttt{=:})\ m$.

There are no suffixes for the `=:` operator, but suffixes for the other two as described below allow some control over the tradeoff among code size, speed of execution, and compilation time.

Semantics

These operators have related meanings. The semantics for the arities not mentioned below follows from the algebraic properties above.

- An expression of the form $\texttt{<}x_0 \ldots x_n\texttt{>-\$<}y_0 \ldots y_n\texttt{>}$ with the left and right operand being lists of equal length, evaluates to a function f such that $f(x_i) = y_i$ for all $0 \leq i \leq n$. The effect of applying f to other arguments than those listed is unspecified and can cause an exception.
- An expression of the form $\texttt{<(}x_0\texttt{,}y_0\texttt{)}\ \ldots\ \texttt{(}x_n\texttt{,}y_n\texttt{)>-:}d$, where d is a function, evaluates to a function f such that $f(x_i) = y_i$ for all $0 \leq i \leq n$, and $f(z) = d(z)$ for all z not in $\{x_0 \ldots x_n\}$.
- An expression of the form $\texttt{-:}\ \ \texttt{<(}x_0\texttt{,}y_0\texttt{)}\ \ldots\ \texttt{(}x_n\texttt{,}y_n\texttt{)>}$ evaluates to a function f such that $f(x_i) = y_i$ for all $0 \leq i \leq n$, and $f(z)$ is undefined for all z not in $\{x_0 \ldots x_n\}$.
- An expression of the form $\texttt{<(}x_0\texttt{,}y_0\texttt{)}\ \ldots\ \texttt{(}x_n\texttt{,}y_n\texttt{)>=:}$ (with no right operand) evaluates to a function f such that $f(x_i) = y_i$ for all $0 \leq i \leq n$ but otherwise is undefined, provided that x_i is an address (of type `%a`) for all i, and all x_i have the same weight.

The address map operator `=:` generates faster code than the others where applicable by exploiting the concrete representation of pointers, provided that the pointers are distinct and non-overlapping.

All of these operators require mutually distinct x values or the results are undefined. However, the y values need not be mutually distinct. If there are many cases of multiple x values mapping to the same y, the code may be optimized automatically to avoid containing redundant copies of y values if doing so results in a net improvement.

Listing 6.2 nested conditional equivalent to Listing 6.1

```
digitize =

conditional(
   compose(compare,couple(constant 0,field &)),
   constant `0,
   conditional(
      compose(compare,couple(constant 1,field &)),
      constant `1,
      conditional(
         compose(compare,couple(constant 2,field &)),
         constant `2,
         conditional(
            compose(compare,couple(constant 3,field &)),
            constant `3,
            conditional(
               compose(compare,couple(constant 4,field &)),
               constant `4,
               conditional(
                  compose(compare,couple(constant 5,field &)),
                  constant `5,
                  conditional(
                     compose(compare,couple(constant 6,field &)),
                     constant `6,
                     constant `7)))))))
```

Tradeoffs

Reifications of large data sets can be time consuming to construct. The time to construct them might outweigh the time saved over a less efficient equivalent. For example, building a cumulative conditional on the fly can be very easily done by a function like this one,

```
h = @p =>0 ~&r?\!@lr ?^(@ll //==,^/!@lr ~&r)
```

which can applied to the pair `((<0,1,2,3,4,5,6,7>,'01234567')` to generate the code shown in Listing 6.2. The resulting function requires an average of 27.2 reductions[11] each time it is evaluated (assuming uniformly distributed inputs), whereas the code in Listing 6.1 requires only 8.2. However, the code in Listing 6.2 requires only 325 reductions to construct from the given data, whereas the alternative requires 11,971.

If the reification is performed only at compile time and the function is used only at run time, there is no issue, but otherwise some experimentation may be needed to find the optimum tradeoff.

[11] A primitive virtual machine operation as measured by the `profile` combinator or compiler directive is called a reduction. Reductions are not quite constant time operations but are close enough for this sort of analysis.

Listing 6.3 a space-optimized reification semantically equivalent to Listings 6.1 and 6.2.

```
$ fun --m="-:=@p (<0,1,2,3,4,5,6,7>,'01234567')" --decompile
main = couple(
   couple(
      constant 0,
      conditional(
         field &,
         conditional(
            field(0,&),
            conditional(
               field(0,(&,0)),
               couple(
                  conditional(field(0,(0,&)),constant `Q,constant -1),
                  field(&,0)),
               couple(
                  constant -1,
                  conditional(field(&,0),constant 1,constant <0,0>))),
            constant(1,<<0,0>>)),
         constant(1,-1)))
```

Suffixes

The default behavior of the `-:` and `-$` operators without a suffix is to generate the code as quickly as possible, by limiting the results to functions that can be constructed from `conditional`, `field`, and `constant` virtual machine combinators. Alternative behaviors can be specified using suffixes of `-` and `=`. The suffixes are mutually exclusive, and have these interpretations.

- `-` requests code that may have better run time performance (in real time rather than number of virtual machine reductions) by factoring out common compositions where possible
- `=` requests code that is as small as possible, by considering more general forms and searching exhaustively

The `=` suffix will incur exponential compilation time, making it infeasible except in special circumstances, but the result will be tighter than humanly possible to write manually. For example, we can obtain a result like Listing 6.3 rather than the code in Listing 6.1 with an improvement in size to 77 quits (down from 106), but the number of reductions required to generate it is 226,355,162 (as opposed to 11,971).

6.11.4 String handlers

The last three operators listed in Table 6.11 are useful for string manipulation, but they also generalize to lists of any type. The `%=` operator is suitable for string substitution, and the `=]` and `[=` operators are for detecting prefixes of strings, which is relevant to parsing and file handling applications.

String substitution

The `%=` operator can be used in all four arities and is fully dyadic. An expression of the form s`%=`t, where s and t are strings (or lists of any type) denotes a function that searches its argument for occurrences of s as a substring and returns a modified copy of the argument in which the occurrences of s have been replaced by t.

Suffixes This operator allows a suffix consisting of any sequence of the characters $\star$, =, and −. The effects of these characters in a suffix can be specified in terms of other operators described in this chapter. When a suffix contains more than one of them, they apply cumulatively in the order they're written.

- The $\star$ used as a suffix makes the result apply to all items of a list.

$$s\texttt{\%=}\star t \;\equiv\; \texttt{(}s\texttt{\%=}t\texttt{)}\,\star$$

- The = as a suffix calls for a postprocessor to flatten the result to its cumulative concatenation.

$$s\texttt{\%==}t \;\equiv\; \texttt{--:-<>+}\;\; s\texttt{\%=}t$$

- The − suffix makes the function iterate as many times as necessary to replace new occurrences of the pattern s that may be created as a consequence of substitutions.

$$s\texttt{\%=-}t \;\equiv\; \texttt{(}s\texttt{\%=}t\texttt{)\^{}=}$$

Prefix recognition

The two remaining operators are `[=` and `=]`, called "prefix" and "startswith", respectively (despite other uses of the word "prefix" in this manual). Both of these operators can be used in any arity, and are postfix dyadic. The left operand, if any, is a function, and the right operand, if any, is a string or a list. They share the algebraic property

$$\texttt{[=}x \;\equiv\; \texttt{\~{}\&[=}x$$

which is to say that the prefix arity is equivalent to the infix arity with an implied left operand of the identity function. Their algebraic properties differ with regard to the solo arity, in that $(\texttt{=]})\ x \;\equiv \texttt{=]}\,x$ whereas $(\texttt{[=})\ (x, y) \;\equiv\; (\texttt{[=}y)\ x$. Neither operator has any suffixes. Their semantics can be summarized as follows.

- The expression $(f\,\texttt{[=}x)\ y$ is true when $f(y)$ is a prefix of x.
- The expression $(f\texttt{=]}\,x)\ y$ is true when x is a prefix of $f(y)$.

The prefixes of a string y are the solutions x to $y = x\texttt{--}z$ with z unconstrained.

	meaning	illustration		
^	coupling	^(f,g) x	≡	(f x,g x)
+	composition	f+g x	≡	f g x
~	deconstructor functional	~p	≡	field p
/	binary to unary combinator	f/k x	≡	f(k,x)
\	reverse binary to unary combinator	f\k x	≡	f(x,k)
!	constant functional	x! y	≡	x
?	conditional	~&w?(~&x,~&r)	≡	~&wxrQ
.	composition or lambda abstraction	~&h.&l	≡	~&hl
*	map	f* <a,b>	≡	<f a,f b>
*~	filter	~=`x*~ 'axbxc'	≡	'abc'
-=	membership	f-= s	≡	~&w^(f,s!)
==	comparison	f== x	≡	~&E^(f,x!)
;	reverse composition	g;f x	≡	f g x
:	list or assignment construction	a:<b>	≡	<a,b>
--	concatenation of lists	<a,b>--<c,d>	≡	<a,b,c,d>
$	record lifter	rec$[a: f,b: g]	≡	^(f,g)
->	iteration	p->f	≡	p?(p->f+ f,~&)
-<	sort	nleq-< <2,1,3>	≡	<1,2,3>

Table 6.12: operator survival kit

6.12 Remarks

The best way to proceed after a first reading of this chapter is to select a subset of the operators such as the one shown in Table 6.12 for use in your initial coding efforts. As the work progresses, you might gradually add to your repertoire when a new challenge can be met most effectively by deploying a new operator.

Despite the importance of this material, attempting to commit it to memory is not recommended.[12] Subtle lapses about semantics or algebraic properties will invariably occur that become persistent habits and code maintenance problems.

The recommended way of staying on top of this material is to make full use of the interactive help facilities of the compiler. Brief reminders of the information in this chapter are at your fingertips during development by way of various interactive commands. For example, to see a complete list of all infix operators with a short reminder about how they work, execute the command

```
$ fun --help infix
```

Similar commands can be used for prefix, postfix, and solo operators. To get help for an individual operator, use a command like this.

```
$ fun --help infix,"->"
```

[12]If the evil day should ever arrive that a job seeker is asked picky questions about this language in an interview, he or she should feel free to quote chapter and verse from this section.

```
infix operators
---------------
->  p->f iterates f while p is true
```

If an operator contains the = character, it may be necessary to invoke the command with this syntax to avoid misleading the command line option parser in the virtual machine.

```
$ fun --help=prefix,"-="
```

Finally, summary information about operator suffixes can be retrieved interactively by the command

```
$ fun --help suffixes
```

This command can also be used for specific operators in the manner described above.

Let's get this freak show on the road.

Sheriff Wydell in *The Devil's Rejects*

7 Compiler directives

A sequential reading of this manual imparts a knowledge of the language from the bottom up, starting with the major components of pointers, types, and operators. Some features remain to be discussed at this point with a view to assembling them into complete applications. This chapter gives a systematic account of the large scale organization of a source text, and is concerned mainly with the use of compiler directives.

7.1 Source file organization

A file containing source code suitable for compilation, usually named with a suffix `.fun`, follows a pattern of sequences of declarations nested within matched pairs of compiler directives. A partial EBNF (Extended Backus-Nauer form) syntactic specification may be useful as a road map.

$$
\begin{aligned}
\langle \textit{source file} \rangle &::= \langle \textit{directive} \rangle (\texttt{+} \mid \langle \textit{expression} \rangle) \\
& [\langle \textit{declaration} \rangle \mid \langle \textit{source file} \rangle] * \\
& \langle \textit{directive} \rangle \texttt{-} \\
\langle \textit{directive} \rangle &::= \texttt{\#} \langle \textit{identifier} \rangle \\
\langle \textit{declaration} \rangle &::= \langle \textit{handle} \rangle \texttt{=} \langle \textit{expression} \rangle \mid \langle \textit{record declaration} \rangle \\
\langle \textit{expression} \rangle &::= \langle \textit{identifier} \rangle \mid \\
& [\langle \textit{expression} \rangle]\ \langle \textit{operator} \rangle\ [\langle \textit{expression} \rangle] \mid \\
& \langle \textit{left aggregator} \rangle [\langle \textit{expression} \rangle [\texttt{,} \langle \textit{expression} \rangle] *] \langle \textit{right aggregator} \rangle
\end{aligned}
$$

In keeping with EBNF conventions, most of the punctuation above is metasyntax. Square brackets contain optional content, vertical bars indicate choice, the $*$ indicates zero or more repetitions, and $::=$ defines a rewrite rule. Only the characters set in typewriter font are meant to be taken literally, namely the comma, plus, minus, =, and hash characters above.

- Expressions consist of operators and operands as documented in Chapter 6.
- Aggregators are things like parentheses and braces as documented in Chapter 5.
- Handles appearing on the left of a declaration are a restricted form of expression to be explained shortly.

7.1.1 Comments

Comments can be interspersed with this file format. There are five kinds of comments. New users need to learn only the first one.

- The delimiters `(#` and `#)` may be used in matched pairs to indicate a comment anywhere in a source file (other than within a quoted string or other atomic lexeme, of course), and may be nested.
- A hash character `#` followed by white space or a non-alphabetic character other than a hash designates the remainder of the line as a comment. A backslash at the end of the line may be used as a comment continuation character.
- Four consecutive dashes designate the remainder of the line as a comment, and it may also have a backslash as a comment continuation character at the end.
- Three consecutive hashes, `###`, indicate that the remainder of the file is a comment.
- A pair of hashes, `##`, followed by anything other than a third hash indicates a smart comment, which may be used to "comment out" a section of syntactically correct code.
  - A smart comment between declarations comments out the next declaration.
  - A smart comment appearing anywhere within a pair of aggregate operators comments out the remainder of the expression in which it appears up to the next comma or closing aggregator at the same nesting level.

There used to be a textbook argument against nested comments based on a contrived example, but the consensus may have shifted in recent years. Readers will have to use their own judgment.

These features are intended to make debugging less tedious when it involves frequently commenting and uncommenting sections of code. Smart comments are a particular innovation of the language that can be demonstrated briefly as follows.

```
$ fun --main="<1,2,3>" --cast %nL
<1,2,3>
$ fun --m="<1,2,## 3>" --c
<1,2>
```

task	directives	effects
visibility	#hide+	make enclosed declarations invisible outside unless exported
	#import	make a given list of symbols visible in the current scope
	#export+	allow declarations to be visible outside the current scope
binary file output	#comment	insert a given string or list of strings into output files
	#binary+	dump each symbol in the current scope to a binary file
	#executable	write an executable file for each function in the current scope
	#library+	write a library file of the symbols defined in the current scope
text file output	#cast	display values to standard output formatted as a given type
	#output	write output files generated by a given function
	#show+	display text valued symbols to standard output
	#text+	write printable symbols in the current scope to text files
code generation	#fix	specify a fixed point combinator for solving circular definitions
	#optimize+	perform extra first order functional optimizations
	#pessimize+	inhibit default functional optimizations
	#profile+	add run time profiling annotations to functions
reflection	#preprocess	filter parse trees through a given function before evaluating
	#postprocess	filter output files through a given function before writing
	#depend	specify build dependences for external development tools

Table 7.1: compiler directives by task classification; non-parameterized directives are shown with a + sign

When smart comments are used in a large expression, there is no need to fish for the other end of it to insert the matching comment delimiter, or to be too concerned about whether the commas and the right number of nesting aggregate operators are inside or outside the comment.

7.1.2 Directives

Compiler directives give instructions to the compiler about what should be done with the code it generates from the declarations. Directives can be nested in matched pairs like parentheses, and their effect is confined to the declarations appearing between them. Every source text needs at least some directives in order for its compilation to have any useful effect, but sometimes the directives are implicit or are stipulated by command line options.

Syntactically, a directive begins with a hash character, followed by an identifier. The opening directive of a matched pair is followed either by a plus sign (with no intervening space) or an expression. The closing directive in a pair contains the same identifier terminated by a minus sign. An expression is supplied only for so called parameterized directives.

Some examples of directives noted previously in passing are the `#library+` directive for creating a library file, and the `#executable` directive for creating an executable file. The latter is a parameterized directive and the former isn't. These and the other directives shown in Table 7.1 are documented more specifically in this chapter.

7.1.3 Declarations

Other than compiler directives and comments, the main things occupying a source file are declarations. There are two kinds of declarations, one for records and the other for general data or functions using the = operator. Record declarations are documented comprehensively in Section 4.2 and need not be revisited here. The = operator is used in many previous examples but may benefit from further explanation below.

Motivation

The purpose of declarations is to effect compile-time bindings of values to identifiers, thereby associating a symbolic name with the value. When a declaration of the form $\langle\mathit{name}\rangle$=$\langle\mathit{value}\rangle$ appears in a source text, the name on the left may be used in place of the value on the right in any expression with the same effect (subject to rules of scope to be explained presently). There are several reasons declarations are important.

- Descriptive names are universally lauded as good programming practice. Complicated code is made more meaningful to a human reader when a large expression is encapsulated by a well chosen name.
- Code maintenance is easier and more reliable when a value used throughout the source text needs to be revised and only its declaration is affected.
- The expression on the right of a declaration is evaluated only once during a compilation, regardless of how many times the name is used. Declaring it thereby improves efficiency if it is used in several places.
- Sometimes the names given to values are needed by output generating directives, for example as file names or as names of symbols in a library.

Declaration Syntax

The right side of the = operator in a declaration of the form

$$\langle\mathit{handle}\rangle \;=\; \langle\mathit{expression}\rangle$$

is an expression composed of operators and operands as documented in Chapters 5 and 6. Usually the left side is a single identifier, but in general it may follow this syntax,

$$\begin{array}{rcl}
\langle\mathit{handle}\rangle & ::= & \langle\mathit{identifier}\rangle \mid \texttt{(}\langle\mathit{handle}\rangle\texttt{)} \mid \langle\mathit{handle}\rangle\;\langle\mathit{params}\rangle \\
\langle\mathit{params}\rangle & ::= & \langle\mathit{variable}\rangle \mid \texttt{(}\langle\mathit{params}\rangle\texttt{[,}\langle\mathit{params}\rangle\texttt{]*)} \mid \texttt{<}\langle\mathit{params}\rangle\texttt{[,}\langle\mathit{params}\rangle\texttt{]*>}
\end{array}$$

where a variable is a double quoted string like `"x"` or `"y"`. That is, the identifier may appear with arbitrarily many dummy variable parameters in lists or tuples nested to any depth. This syntax is the same as the part of a record declaration to the left of the `::` operator. (See Section 4.2.4, page 162.) Note that no terminators or separators other than white space are required between declarations.

Interpretation of dummy variables

If dummy variables appear in the handle, the declaration is that of a function and the variables are part of a syntactically sugared form of lambda abstraction (pages 24 and 207). The declaration $(f\ x)\ =\ y$ is transformed to $f\ =\ x\texttt{.}\ \ y$. More generally, a declaration of the form

$$(\dots(f\ x_0)\dots x_n)\ =\ y$$

is transformed to

$$(\dots(f\ x_0)\dots x_{n-1})\ =\ x_n\texttt{.}\ \ y$$

(and so on). Free occurrences of the variables may appear in the expression y.

Identifier syntax

Identifiers abide by the following syntactic rules.

- An identifier may consist of upper and lower case letters and underscores, but not digits. This convention allows functions and numerical arguments to be juxtaposed without spaces or parentheses, with an expression like `h1` being parsed as `h(1)`.
- The letters in an identifier are case sensitive, so `foobar` is a different identifier from `FooBar`.
- Identifiers beginning with underscores may not be declared, because they are reserved either for record type expression identifiers or for a very few predeclared identifiers.
- Identifiers for compiler directives and standard library functions are not reserved, making it acceptable to redefine words like `library` and `conditional`.

Predeclared identifiers

Predeclared identifiers begin with two underscores, and there are currently only a small number of them. They are provided as predeclared identifiers rather than library functions for obvious reasons demanded by their semantics.

- `__switches` evaluates to a list of strings given by the command line parameters to the `--switches` option when the compiler is invoked.
- `__ursala_version` evaluates to a character string giving the version number of the compiler.
- `__source_time_stamp` evaluates to a character string containing the modification date and time of the source file in which it appears.

The `__switches` feature allows the code to be dependent in arbitrary ways on user-defined compile-time flags. Typical applications would be to enable or disable profiling or assertions, and for conditional compilation of platform dependent code.

For example, a development version of an application may need to use the `profile` combinator to generate run time statistics so that the hot spots can be identified and optimized, but the production version can exclude it. (See the `avram` reference manual for more information about profiling.) This declaration appearing in the source

```
profile = -=/'profile'?(std-profile!,~&l!) __switches
```

will redefined the `profile` combinator as a no-op unless

```
--switches=profile
```

is used as a command line option during compilation. Note that the choice of the word "`profile`" as a switch is arbitrary and independent of the standard function by the same name (or for that matter, the compiler directive with the same name).

7.2 Scope

Rules of scope are rarely a matter of concern for a user of this language, because the conventions are intuitive. Normally an identifier declared in a source file can be used anywhere else in the same file, before or after the declaration. Multiple declarations of the same identifier are an error and will cause compile time exception. Identifiers declared in separately compiled files are stored in libraries that may be imported. Applications for which these arrangements are insufficient are probably over designed.

Nevertheless, there are ways of deliberately controlling the scope and visibility of declarations using the first three compiler directives listed in Table 7.1, which are documented in this section.

7.2.1 The `#import` directive

Almost every source file contains `#import` directives in order to make use of standard or user defined libraries.

- The `#import` directive is parameterized by an expression whose value is a list of assignments of strings to values, that may optionally be compressed (i.e., type `%om` or `%omQ` in terms of type expressions documented in Chapter 3).
- The effect of the `#import` directive on an expression `<'foo': bar, ...>` is similar to inserting the sequence of declarations `foo = bar...` at the point in the file where the directive is invoked.
- A matching `#import-` directive may appear subsequently in the file, but has no effect.

Usage

Many previous examples have featured the directives

```
#import std
#import nat
```

for importing the standard library and natural number library. This practice is effective because external libraries are stored in binary files as instances of `%om` or `%omQ`, and any binary file name mentioned on the command line during compilation is accessible as an identifier in the source. However, nothing prevents arbitrary user defined expressions of these types from being "imported". (The `std` and `nat` libraries don't have to be named on the command line because they are automatically supplied by the shell script that invokes the compiler.)

Semantics

The effect of an `#import` directive is similar but not identical to inserting declarations. Although it is normally an error to have multiple declarations of the same identifier, it is acceptable to have a locally declared identifier with the same name as one that is imported. In this case, the local declaration takes precedence, but the precedence can be overridden by the dash operator.

It is also acceptable to import multiple libraries with some identifiers in common. In this case, it is best to use fully qualified names with the dash operator (Section 6.7.1, page 215). For example, if two libraries `foo` and `bar` both need to be imported and both include an identifier `x`, then uses of `x` in the source should be qualified as `foo-x` or `bar-x` as the case may be.

Name clashes Although relying on it would be asking for maintenance problems, there is a rule for name clash resolution when multiple libraries containing the same symbol name are imported.

- The library whose importation most recently precedes the use of an identifier in the text takes precedence.
- If all relevant importations follow the use of an identifier in the text, the last one takes precedence.

Type expressions The compiler uses a compressed format for the concrete representations of type expressions in library modules that differs from their run-time representations. The `#import` directive treats the value of an identifier beginning with an underscore as a type expression and transparently effects the transformation, based on the assumption that these identifiers are reserved for type expressions. If a type expression is invalid, an exception occurs with the diagnostic message "`bad #imported type expression`". A deliberate effort would be required to cause this exception.

7.2.2 The `#export+` directive

The main use for this directive is in a situation where dependences exist in both directions between declarations in separate source files. This situation makes it impossible to compile one of them first into a library and then import it by the other.

Motivation

This situation is avoidable. Assuming no dependence cycles exist between declarations, the problem could be solved by merging or reorganizing the files. (For coping with cyclic dependences, see the `#fix` directive later in this chapter.) However, if design preferences are otherwise, the user can also arrange to compile both source files simultaneously without merging them just by naming both on the command line when invoking the compiler.

Simultaneous compilation does not fully resolve the issue in itself. When multiple files are compiled simultaneously, the declarations in one file are not normally visible in another. (I.e., an attempt to use an identifier declared in another file will cause a compile-time exception with an "`unrecognized identifier`" diagnostic message.) However, the `#export+` directive can make declarations visible outside the file where they are written.

Usage

The usage of the `#export` directives is very simple. To make all declarations in a source file visible, place `#export+` near the beginning of the file before any declarations. To make declarations visible only selectively, insert `#export+` and `#export-` anywhere between declarations in the file. Only the declarations that are more recently preceded by `#export+` than `#export-` will then be visible.

Semantics

A couple of points of semantics should be noted.

- The effect of `#export+` is orthogonal to directives that generate output files, such as `#binary+` or `#library+`, which can cause declarations to be written to files whether they are visible or not.
- The `#export` directive can be overridden by the `#hide` directive, and vice versa, as explained in the next section.
- Name clashes are possible when multiple files compiled simultaneously export symbols with the same names.
  - Local declarations take precedence over external declarations.
  - Further rules of name clash priority are given in the next section.
  - An expression like `filename-symbol` can be used similarly to the dash operator to qualify a symbol unambiguously, unless not even the file names are unique.

The last point pertains to an idiom of the language rather than a legitimate use of the dash operator, because the file name is not meaningful as an operand in itself.

7.2.3 The `#hide+` directive

Even further removed from common use is the `#hide+` directive, which can create separate local name spaces within a single source file. Although it is unlikely to be needed by a real user, this directive is used internally by the compiler, making it a feature of the language calling for documentation. In particular, the name clash priority rules for simultaneously compiled files are implied by its specification, with a matched pair of these directives implicitly bracketing each source file and another bracketing their ensemble.

Usage

The `#hide+` and `#hide-` directives can be used as follows. Readers who find these matters perfectly lucid probably have been thinking about programming languages too long.

- Unlike other directives, these directives can occur only in properly nested matched pairs, or else an exception is raised.
- The declarations between a pair of `#hide+` and `#hide-` directives are not normally visible outside them, even within the same file.
- The `#export` directives can be used in conjunction with the `#hide` directives to make declarations selectively visible outside their immediate name space.
  - The visibility extends only one level outward by default.
  - A symbol can be exported another level outward by a further `#export+` directive that textually precedes the symbol's enclosing `#hide+` directive at the same level (and so on).
- If no `#export` directives are used within a given name space, then by default the last symbol declared (textually) is visible one level outward.
- If a symbol exported from a nested space (or visible by default) has the same name as a symbol that is exported from a space containing it, only the latter is visible outside the enclosing space.

Name clashes

To complete the picture, a name clash resolution policy is needed when multiple declarations of the same identifier are visible. For this purpose, we can regard name spaces as forming a tree, with nested spaces as the descendents of those enclosing them. The least common ancestor of any two nodes is the smallest subtree containing them.

- The name clash resolution policy favors the declaration of an identifier whose least common ancestor with the declaration using it is the minimum.
- If multiple declarations meet the above criterion, preference is given to the one that textually precedes the use of the identifier most closely, if any.
- If the there are multiple minima and none of them precedes the use, the one closest to the end of the file takes precedence.

The ordering of textual precedence is generalized to multiple files based on their order in the command line invocation of the compiler.

7.3 Binary file output

There are four directives that are relevant to the output of binary files. Library files, executable files, and binary data files are each written by way of a separate directive, and the remaining directive inserts comments into any of these file types.

7.3.1 Binary data files

Any data of any type generated in the course of a compilation can be saved in a file for future use by the `#binary+` directive. The file format is standardized by the compiler and the virtual machine so that no printing or parsing needs to be specified by the user. Although they are called binary files in this manual, they actually contain only printable characters as a matter of convenience. The use of printable characters does not restrict the types of their contents.

Usage

The usual way to generate binary data files is by having a `#binary+` directive preceding any number of declarations, optionally followed by a `#binary-` directive.

$$\begin{array}{lcl} \texttt{\#binary+} & & \\ \langle \textit{identifier} \rangle_1 & = & \langle \textit{expression} \rangle_1 \\ & \vdots & \\ \langle \textit{identifier} \rangle_n & = & \langle \textit{expression} \rangle_n \\ \texttt{\#binary-} & & \end{array}$$

Compilation of this code will cause n binary files to be written to the current directory, with file names given by the identifiers and contents given by the expressions. If the `#binary-` directive is omitted, then all declarations up to the end of the file or the next `#hide-` directive are involved.

Other forms of declarations can also be used to generate binary files, such as records, lambda abstractions, and imported libraries.

- In the case of a record declaration, a separate file will be written for each field identifier, for the record type expression, and for the record initializing function.
- If the left side of a declaration is parameterized with dummy variables, the file is named after the identifier without the parameters, and it contains the virtual machine code for the function determined by the lambda abstraction (page 249).
- If an `#import` directive (Section 7.2.1) appears within the scope of a `#binary+` directive, one file is written for each imported symbol.

It is an error to attempt to cause multiple binary files with the same name to be written in the same directory. There is no provision for name clash resolution, and an exception is raised.

Example

A short example shows how a numerical value can be written to a binary file and then used in a subsequent compilation.

```
$ fun --m="#binary+ x=1"
fun: writing `x'
$ fun x --m=x --c
1
```

The value in a binary file is used by passing the file name as a command line parameter to the compiler, and using the name of the file as an identifier in the source text.

7.3.2 Library files

The `#library+` and `#library-` directives may be used to bracket any sequence of declarations in a source text to store them in a library file, as shown below.

$$
\begin{array}{lcl}
\texttt{\#library+} & & \\
\langle \textit{identifier} \rangle_1 & = & \langle \textit{expression} \rangle_1 \\
 & \vdots & \\
\langle \textit{identifier} \rangle_n & = & \langle \textit{expression} \rangle_n \\
\texttt{\#library-} & &
\end{array}
$$

If the `#library-` directive is omitted, the scope of the `#library+` directives extends to the end of the file or current name space. The declarations can also be for imported modules or records.

Usage

The binary file written in the case of the `#library+` directive is named after the source file in which it appears, with a suffix of `.avm`. At most one library file is written for each

Listing 7.1 a library source file

```
#library+

rec :: x y

foo = 'a
bar = 'b
baz = 'c
```

Listing 7.2 excerpt of the binary file from Listing 7.1

```
# rec                                                    (9)
#    - x
#    - y
# bar                                                    (6)
# baz                                                    (7)
# foo                                                    (5)
#
{w{yZKk`{AsMU{r[yU[sx\Mz[MAnkczDqmAac\AlZ[_[ra<MeUxKbKYop^D`Et[?JxPQ...
Sh{^`wKtuzD]ZozD]Z\=XJ[^DS_ctcd<S?cv<Ar]^Z\=XEt=VBEz]d=VB<L\@^<
```

source file. If multiple pairs of `#library+` and `#library-` directives appear in a file, all of the declarations between each pair are collected together into the same file.

The normal way to use a library file is by the `#import` directive, which will cause the symbols stored in the library to be declared in the current name space, as explained in Section 7.2.1. A library file can also be used directly as a list of assignments of strings to values (type `%om`) or as a compressed list of assignments of strings to values (type `%omQ`). A library will be compressed if the command line option `--archive` is used when it is compiled.

Example

An example of a library file is shown in Listing 7.1, and part of the binary file is shown in Listing 7.2.

File formats The binary file for a library contains an automatically generated preamble listing the symbols alphabetically and their sizes measured in two bit units (quits). If any records are declared in the library, they are listed first with the field identifiers as shown. This format makes it easy to find the file containing a known symbol in a directory of library files by a command such as the following.

```
$ grep foo *.avm
libdem.avm:# foo                                          (5)
```

Compilation The library source file is compiled by the command

```
$ fun libdem.fun
fun: writing `libdem.avm'
```

It can be tested as follows.

```
$ fun libdem --main="<foo,bar,baz>" --cast
'abc'
```

The suffix `.avm` on the file name may be omitted when the file name is given as a command line parameter. When library symbols are referenced in a `--main` expression, no `#import` directive is necessary, but if the library were used in a source file, the `#import libdem` directive would be needed in the file.

7.3.3 Executable files

An executable file is one that can be invoked as a shell command to perform a computation. The compiler can be used to generate executable files from specifications in Ursala, which are implemented as wrapper scripts that launch the virtual machine (`avram`) loaded with the necessary code. These scripts appear to execute natively to the end user, but are portable to any platform on which the virtual machine is installed.

Usage

The `#executable` directive is used to generate executable files. It is normally appears in a source text as shown.

$$\begin{array}{lcl} \texttt{\#executable} & \multicolumn{2}{l}{\texttt{(}\langle\textit{options}\rangle\texttt{,}\langle\textit{configuration files}\rangle\texttt{)}} \\ \langle\textit{identifier}\rangle_1 & \texttt{=} & \langle\textit{expression}\rangle_1 \\ & \vdots & \\ \langle\textit{identifier}\rangle_n & \texttt{=} & \langle\textit{expression}\rangle_n \\ \texttt{\#executable-} & & \end{array}$$

The options and configuration files are lists of strings, which may be empty.

- The idiomatic usage `#executable&` pertains to an executable with no options and no configuration files.
- Each enclosed declaration should represent a function that is meaningful to invoke as a free standing application.
- If the `#executable-` directive is omitted, all declarations up to the end of the current name space are included.
- A separate executable file is written for each declaration, named after the identifier.

Execution models

The run time behavior of an executable file is specified partly by the function it contains and partly by the way the virtual machine is invoked. The latter is determined by the options given in the left side of the parameter to the `#executable` directive, which are supplied automatically to the virtual machine as command line options.

A complete list of command line options for the virtual machine with brief explanations can be viewed by executing the command

```
$ avram --help
```

All options are documented extensively in the `avram` reference manual. Some of them are less frequently used because they are applicable only in special circumstances, such as infinite stream processing, but the two that suffice for most applications are the following.

- A directive of the form

  `#executable (<'parameterized'>,`⟨*configuration files*⟩`)`

  will cause the virtual machine to pass a data structure containing the environment variables, file parameters, and command line options as an argument to the function declared under it. The function will be required to return a list of data structures representing files, which will be written to the host's file system by the virtual machine.

- A directive of the form

  `#executable (<'unparameterized'>,<>)`

  will cause the virtual machine to pass a list of character strings to the function declared under it, which are read from the standard input stream at run time, up to the end of the file. The function will be required to return a list of character strings, which the virtual machine will write to standard output. Configuration files are not applicable to this usage.

These options may be recognizably truncated, for example as `'p'`, and `'u'`. The latter is assumed by default if no options are specified and the executable is invoked at run time with no command line parameters. Nothing more needs to be said about unparameterized execution, but the alternative is documented below.

Parameterized execution

The main argument to a function compiled to an executable file using the `'par'` option is a record of type `_invocation`, as defined by the standard library distributed with the compiler and excerpted in Listing 7.3. This record is initialized by the virtual machine at run time depending on how the executable is invoked. Familiarity with the conventions pertaining to record declarations and usage documented in previous chapters would be helpful for understanding this section.

Listing 7.3 data structures used by parameterized executable files

```
command_line    :: files _file%L options _option%L
file            :: stamp %sbU path %sL preamble %sL contents %sLxU
option          :: position %n longform %b keyword %s parameters %sL
invocation      :: command _command_line environs %sm
```

Listing 7.4 a utility to display the command line record

```
#import std

#comment -[
Invoked with any combination of parameters or options,
this program pretty prints a representation of the command line
record to standard output.]-

#executable ('parameterized',<>)

#optimize+

crec = ~&iNC+ file$[contents: --<''>+ _command_line%P+ ~command]
```

Invocation records There are two fields in an `invocation` record, one for the environment variables, and the other for the command line parameters and options.

- The environment variables are represented in the `environs` field as a list of assignments of environment variable identifiers to strings, such as

  `<'DISPLAY': ':0.0','VISUAL': 'xemacs' ...>`

  These are the usual environment variables familiar to Unix and GNU/Linux developers and users, which are initialized by the `set` or `export` shell commands prior to execution.

- The `command` field is a record of type `_command_line`, with two fields, one containing a list of the file parameters and the other containing a list of the command line options.

Some applications might not depend on the environment variables and will be expressed as something like `my_app = ~command; ...`. The rest of the code in an expression of this form accesses only the command line record.

Command line records The data structures used to represent files and command line options are designed to allow convenient access with mnemonic field identifiers. As an example, a short text file

```
$ cat mary.txt
Mary had a little lamb.
```

passed as a command line argument to the application shown in Listing 7.4 with some other parameters will have the output below.

```
$ crec mary.txt --foo --bar=baz
command_line[
   files: <
      file[
         stamp: 'Sun Apr 29 13:48:48 2007',
         path: <'mary.txt'>,
         contents: <'Mary had a little lamb.',''>]>,
   options: <
      option[position: 1,longform: true,keyword: 'foo'],
      option[
         position: 2,
         longform: true,
         keyword: 'bar',
         parameters: <'baz'>]>]
```

The application in Listing 7.4 is distributed with the compiler under the `contrib` subdirectory.

- The `files` field in a command line record contains the list of files separately from the `options` field in the order the files are named on the command line.
- If any configuration file names are supplied to the `#executable` directive when the application is compiled, their files will appear at the beginning of the list without the end user having to specify them.
- The application aborts if any file parameters or configuration files don't exist or aren't readable.

File records The records in the list of files stored in the command line record passed to an application are organized with four fields.

- The `stamp` field contains the modification time of an input file expressed as a string, if available.
- The `path` field is a list of strings whose first item is the file name. Following strings, if any, are parent directory names in ascending order. If the last string in the list is empty, the path is absolute, but otherwise it is relative to the current directory. An empty path refers to the standard input stream.
- The `preamble` is a list of character strings that is empty for text files an non-empty for binary files. Any comments or other front matter stored in a binary file are recorded here.
- The `contents` field is a list of character strings for text files and any type for binary files.

As mentioned previously, file records are also used for output. When an application returns a list of files for output, similar conventions apply except as follows.

- The `stamp` field is treated as a boolean value. If it is non-empty, any existing file at the given path is overwritten, but if it is empty, the file is appended.
- An empty path in an output file record refers to standard output rather than standard input.

There is no direct control over the attributes of output files, but any binary file whose preamble's first line begins with `!` will be detected by the virtual machine and marked as executable.

Option records The other field in a command line record contains a list of records representing the command line options. This field is initialized by the virtual machine to contain the command line options passed to the application when it is invoked. Although command line options are parsed automatically by the virtual machine, it is the application developer's responsibility to validate them.

An option record contains four fields and their interpretations are straightforward.

- The `position` field is a natural number whose value implies the relative ordering of the options and file parameters. This information is useful only to applications whose options have position dependent semantics. Positions are numbered from the left starting at zero. Non-consecutive position numbers between consecutive options indicate intervening file parameters.
- The `longform` field is true if the option is specified with two dashes, and false otherwise.
- The `keyword` field contains the literal name of the option as given on the command line in a character string.
- The `parameters` field contains any associated parameters following the option with an optional = in a comma separated list.

Some experimentation with the `crec` application (Listing 7.4) may be helpful for demonstrating these conventions.

Interactive applications

Applications that perform interactive user input are not unmanageable in Ursala but they may constitute a duplication of effort. The major classes of applications that need to be interactive, such as editors, browsers, image manipulation programs, *etcetera*, contain mature representatives with robust, extensible designs allowing new modules or plugins. One of them undoubtedly would be the best choice for the front end to any interactive application implemented in this language. It should also be mentioned that functional

Listing 7.5 An application to perform interactive user input

```
#import std
#import cli

#executable (<'par'>,<>)

grab =

~&iNC+ file$[
   stamp: &!,
   path: <'transcript'>!,
   contents: --<''>+ ~&zm+ ask(bash)/<>+ <'zenity --entry'>!]
```

languages are notoriously awkward at user interaction despite long years of effort by the community to put the best face on it.

With this disclaimer, one small example of an interactive application is shown in Listing 7.5. This application opens a dialog window in which the user can type some text. When the user clicks on the "ok" button, the window closes, and the application writes the text to the a file named `transcript` in the current directory.

The application can be compiled and run as shown below. Although the dialog window isn't shown, that's where the text was entered.

```
$ fun cli grab.fun
fun: writing `grab'
$ grab
grab: writing `transcript'
$ cat transcript
this text was entered
```

The real work is done by the `zenity` utility, which needs to be installed on the host system. It is invoked in a shell spawned by the `ask` function defined in the `cli` library, as documented in Part III of this manual.

7.3.4 Comments

The `#comment` directive adds user supplied front matter to binary data files, libraries, and executable files without altering their semantics. It requires a parameter that is either a character string or a list of character strings.

The text of the comment can be anything at all, and is normally something to document the file for the benefit of an end user. Instructions for an executable or calling conventions for a library file are appropriate. Comments are also good places to include version information obtained by the pre-declared identifiers `__source_time_stamp` or `__ursala_version` (page 249).

A pair of comment directives must bracket the directives that generate the files in which comments are desired. The closing `#comment-` directive may be omitted, in which case the effect extends to the end of the enclosing name space (normally the end of the source file unless `#hide` directives are in use). A general outline of a source file using `#comment` directives would be the following.

$$
\begin{array}{l}
\texttt{\#comment}\ \langle \textit{text} \rangle \\
\\
\langle \textit{directive} \rangle \texttt{(+|}\langle \textit{expression} \rangle\texttt{)} \\
\langle \textit{declaration} \rangle \\
\vdots \\
\langle \textit{declaration} \rangle \\
\langle \textit{directive} \rangle \texttt{-} \\
\vdots \\
\langle \textit{directive} \rangle \texttt{(+|}\langle \textit{expression} \rangle\texttt{)} \\
\langle \textit{declaration} \rangle \\
\vdots \\
\langle \textit{declaration} \rangle \\
\langle \textit{directive} \rangle \texttt{-} \\
\\
\texttt{\#comment-}
\end{array}
$$

As the above syntax suggests, a single comment directive may apply to multiple binary file generating directives, each of which may apply to multiple declarations. The same comment will be inserted into every file that is generated.

More complicated variations on this usage are possible by having nested pairs of comment directives. The outer comment will be written to every output file, and the inner ones will be written in addition only to files generated by the particular directives they bracket.

Although it is intended primarily for binary files, the `#comment` directive can also be used in conjunction with the `#text` and `#output` directives documented in the next section. In these cases, it is the user's responsibility to ensure that the comment does not interfere with the semantic content of the files.

7.4 Text file output

There are four directives pertaining to the output of text files, as shown in Table 7.1. The `#cast` and `#output` are parameterized, whereas `#show+` and `#text+` directives are not. All of them may be used in matched pairs to bracket a sequence of declarations, and will apply only to those they enclose. If the matching member of the pair is omitted, their scope extends to the end of the file or current name space. The specific features of each directive are documented in the remainder of this section.

7.4.1 The `#cast` directive

The `#cast` directive requires a type expression as a parameter, and applies to declarations of values that are instances of the type. It ignores all but the last declaration within the sequence it brackets, and causes the value of the last one to be displayed on standard output. The display follows the concrete syntax implied by the type expression.

This directive therefore performs the same operation as the `--cast` command line option used in many previous examples, except that it occurs within the file instead of on the command line, and the type expression is not optional.

7.4.2 The `#show+` directive

The `#show+` directive performs a similar operation to the `#cast`, explained above, except that no type expression or any other parameter is required. It ignores all but the last declaration in the sequence it brackets, and causes the last one to be written to standard output. The type of the value that is written must be a list of character strings, or else an exception is raised. No formatting of the data is performed.

The `#show+` directive performs the same operation as the `--show` command line option, except that it occurs within the source text instead of on the command line.

7.4.3 The `#text+` directive

This directive causes a text file to be written for each declaration within its scope. The text file is named after the identifier on the left side of the declaration, with a suffix of `.txt` appended. The value of the expression on the right is required to be a list of character strings, but if the value is of a different type, the declaration is silently ignored and no exception is raised. A short example using this directive is the following.

```
$ fun --m="#text+ foo = <'bar',''>"
fun: writing `foo.txt'
$ cat foo.txt
bar
```

7.4.4 The `#output` directive

This directive allows more control over the names and contents of output files than is possible with other directives. It is parameterized by a function whose input is a list of assignments of character strings to values, and whose output is a list of file records as documented on page 260.

Interface

The input to the function parameterizing the `#output` directive contains the values and identifiers of the declarations in its scope, as this example demonstrates.

```
$ fun --m="#output %nmM foo=1 bar=2"
fun:command-line: <'foo': 1,'bar': 2>
```

The error messenger `%nmM` reports its argument in a diagnostic message when control passes to it, as documented on page 145. The argument of `<'foo': 1,'bar': 2>` is derived from the declarations following the directive.

The output from the function may make any use at all of the input or ignore it entirely when generating the list of files to be written, as the next example shows.[1]

```
$ fun --m="#output <file[contents: <'done',''>]>! foo=1"
done
```

- There is the option of defining a non-empty preamble field to generate a binary file rather than a text file.
- A non-empty path will cause the output to be written to a file rather than to standard output.
- Arbitrary binary data can be written in text files by using non-printing characters. A byte value of n is written for the n-th item in `std-characters`.

Alternative interface

It is often more convenient to use the `#output` directive with the function `dot`, which the standard library defines as follows.

```
"s". "f". * file$[
   stamp: &!,
   path: ~&iNC+ --(:/`. "s")+ ~&n,
   contents: "f"+ ~&m]
```

The `dot` function is used in a directive of the form

`#output dot`⟨*suffix*⟩ ⟨*function*⟩

which causes a separate file to be written for each declaration within the scope of the directive. The file is named after the identifier in the declaration with the suffix appended, and the contents of the file are computed by applying the function to the value of the declaration. The function is required to return a list of character strings.

7.5 Code generation

Several directives modify the code generated by the compiler with regard to optimization, profiling, and handling of cyclic dependences. The last requires some discussion at length, but the others are easily understood.

[1]The shell command `set +H` may be needed in advance to suppress interpretation of the exclamation point.

7.5.1 Profiling

The virtual machine provides the means to profile an application by making a record of its run time statistics. For any profiled function, the number of times it is evaluated is tabulated, along with the total and average number of virtual machine instructions (a.k.a. reductions) required to evaluate it, and their percentage of the total. This information may be useful for a developer to identify performance bottlenecks and potential areas for performance tuning.

Profiling a function does not alter its semantics or behavior in any way. The run time statistics are recorded in a file named `profile.txt` in the current directory, without affecting any other file operations.

One way of profiling a function `f` is to substitute the function `profile(f,s)` for it, where `s` is a character string used to identify `f` in the table of profile statistics, and `profile` is a function provided by the standard library. However, it may sometimes be more convenient to use the `#profile+` directive.

Usage

When a sequence of declarations is enclosed within a pair of `#profile` directives, profiling is enabled for all of them. A simple example demonstrates the effect.

```
$ fun --m="#profile+ f=~& #profile- x = f* 'abc'" --c
'abc'
$ cat profile.txt

 invocations    reductions       average    percentage

           3             3           1.0         0.000  f
           1      18522430    18522430.0       100.000

18522433 reductions in total
```

The table shows that `f` was invoked three times, each invocation required one reduction, and these three reductions were approximately zero percent of the total number of reductions performed in the course of compilation and evaluation. These statistics are consistent with the fact that `f` was mapped over a three item list, and its definition as the identity function makes it the simplest possible function.

Hazards

The `#profile` directives are simple to use, but care must be taken to apply them selectively only to functions and not to general data declarations, which they might alter in unpredictable ways. In the above example, profiling is specifically switched off so as not to affect the declaration of `x`, which is not a function. Otherwise we would have this anomalous result.

```
$ fun --m="#profile+ f=~& g=f* 'abc'" --c
(&,&,0,<('abc','g')>)
```

As one might imagine, overlooking this requirement can lead to mysterious bugs.

Another hazard of the `#profile` directives is their use in combination with higher order functions. Although it is not incorrect to profile a higher order function, it might not be very informative. In this code fragment,

```
#profile+
(h "n") "x" = ...
#profile-
t = h1 x
u = h2 x
```

only the function `h` is profiled, which is a higher order function taking a natural number to one of a family of functions. However, the statistics of interest are likely to be those of `h1` and `h2`, which are not profiled. Extending the scope of the `#profile` directives would not address the issue and in fact may cause further problems as described above. This situation calls for using the `profile` function mentioned previously for more specific control than the `#profile` directives.

7.5.2 Optimization directives

A tradeoff exists between the speed of code generation and the quality of the code based on its size and efficiency. For production code, the quality is more important than the time needed to generate it. For code that exists only during the development cycle, the speed of generating the code is advantageous. By default, a middle ground between these alternatives is taken, but it is possible to direct the compiler to make the code more optimal than usual, or to make it less optimal but more quickly generated.

Examples

The directive to improve the quality of the code is `#optimize+`, and the directive to improve the speed of generating it is `#pessimize+`. The first can be demonstrated as follows.

```
$ fun --m="f=%bP" --decompile
f = compose(
   couple(
      conditional(
         field(0,&),
         constant 'true',
         constant 'false'),
      constant 0),
   couple(constant 0,field &))
```

The above code is compiled without optimization, but an improved version is obtained when optimization is requested.

```
$ fun --m="#optimize+ f=%bP" --decompile
f = couple(
   conditional(field &,constant 'true',constant 'false'),
   constant 0)
```

Some understanding of the virtual machine semantics may be needed to recognize that these two programs are equivalent, but it should be clear that the latter is smaller and faster. The `#pessimize+` directive is demonstrated on a different example.

```
$ fun --m="f = ~&x+~&y" --decompile
f = compose(field(0,&),reverse)
$ fun --m="#pessimize+ f = ~&x+~&y" --decompile
f = compose(
   reverse,
   compose(reverse,compose(field(0,&),reverse)))
```

Although there is no reason to use the `#pessimize` directives in cases like the one above, it often occurs during the development cycle that a short test program takes several minutes to compile because a large library function used in the program is being optimized every time. These delays can be mitigated considerably by the `#pessimize` directives.

Hazards

The same care is needed with the `#optimize` directives as with the `#profile` directives to avoid using them on declarations other than functions, for the reasons discussed above. It is sometimes possible to detect a non-function during optimization, and in such cases a warning is issued, but the detection is not completely reliable.

Pessimization can safely be applied to anything with no anomalous effects. However, it is probably never a good idea to have pessimized code in a library function or executable, so a warning is issued when the `#library` or `#executable` directives detect a `#pessimize` directive within their scope.

7.5.3 Fixed point combinators

The `#fix` directive is an unusual feature of the language making it possible to solve systems of recurrences over any semantic domain to any order. It is necessary only for the user to nominate a fixed point combinator specific to the domain of interest, or a hierarchy of fixed point combinators if solutions to systems in higher orders are desired. Systems of recurrences involving multiple semantic domains are also manageable.

First order recurrences

Recurrences involving functions are the most familiar example, because in most languages there is no alternative for expressing recursively defined functions. Listing 7.6 shows an

Listing 7.6 a naive first order functional fixed point combinator

```
#import std

#fix "h". refer ^H("h"+ refer+ ~&f,~&a)

rev = ~&?\~& ^lrNCT\~&h rev+ ~&t
```

example of a recursively defined list reversal function expressed in this style. To see that it really works, we can save it in a file named `fffx.fun` and test it as follows.

```
$ fun fffx.fun --m="rev 'abc'" --c
'cba'
```

Normally a declaration of a function `rev` defined in terms of `rev` would be circular and compilation would fail, but the fixed point combinator

```
"h". refer ^H("h"+ refer+ ~&f,~&a)
```

tells the compiler how to resolve the dependence.

Calling conventions The calling convention for a first order fixed point combinator (i.e., the function supplied by the user as a parameter to the `#fix` directive) is that given a function h, it must return an argument x such that $x = h(x)$. Intuitively, h can be envisioned as a function that plugs something into an expression to arrive at the right hand side of a declaration. In this example, the function h would be

$$h(x) = \texttt{\~{}\&?\textbackslash\~{}\& \^{}lrNCT\textbackslash\~{}\&h } x\texttt{+ \~{}\&t}$$

In particular, $h(\texttt{rev})$ would yield exactly the right hand side of the declaration in Listing 7.6. Since the right hand side is equal to `rev` by definition, the value of `rev` satisfying $\texttt{rev} = h(\texttt{rev})$ is the solution, if it can be found. The job of the fixed point combinator is to find it, hence the calling convention above.

Semantic note The rich and beautiful theory of this subject is beyond the scope of this manual, but it should be noted that the most natural definition of a fixed point for most functions h of interest generally turns out to be an infinite structure in some form. In practice, a finitely describable approximation to it must be found. It is this requirement that calls on the developer's ingenuity. The fixed point combinator in the above example works by creating self modifying code that unrolls as far as necessary at run time, but this method is only the most naive approach.

The construction of fixed point combinators varies widely with the application domain, thereby precluding any standard recipe. For example, these techniques have been used successfully for solving recurrences over asynchronous process networks in an electronic circuit CAD system, where the fixed point combinator takes a considerably different form. Specific applications are not discussed further here.

Listing 7.7 a better first order functional fixed point combinator

```
#import std
#import sol

#fix function_fixer

rev = ~&?\~& ^lrNCT\~&h rev+ ~&t
```

Practical functional recurrences There are of course better ways of expressing list reversal and recursively defined functions in general. Even for recurrences in this style, the fixed point combinator in Listing 7.6 should never be used in practice because it generates bloated code, albeit semantically correct. Users who are nevertheless partial to this style, perhaps due to prior experience with other languages, are advised to use the `function_fixer` as a fixed point combinator, as shown in Listing 7.7, from the `sol` library distributed with the compiler.

```
$ fun sol bffx.fun --decompile
rev = refer conditional(
   field(0,&),
   compose(
      cat,
      couple(
         recur((&,0),(0,(0,&))),
         couple(field(0,(&,0)),constant 0))),
   field(0,&))
```

The results are seen to be comparable in quality to hand written code, although not as good as using the virtual machine's built in `reverse` function or `~&x` pseudo-pointer.

Higher order recurrences

The recurrences considered up to this point are of the form $t = h(t)$, but there may also be a need to solve higher order recurrences in these forms,

$$\begin{aligned} t &= \texttt{"x0".}\ h(t, \texttt{"x0"}) \\ t &= \texttt{"x0".}\ \texttt{"x1".}\ h(t, \texttt{"x0"}, \texttt{"x1"}) \\ t &= \texttt{"x0".}\ \texttt{"x1".}\ \texttt{"x2".}\ h(t, \texttt{"x0"}, \texttt{"x1"}, \texttt{"x2"}) \\ &\ \ \vdots \end{aligned}$$

and their equivalents, $t(\texttt{"x0"}) = h(t, \texttt{"x0"})$, or variable-free forms $t = h/t$, and so on. In these recurrences, t has a higher order functional semantics regardless of the domain. The order is at least the number of nested lambda abstractions, but could be greater if the expressions are written in a variable-free style. It can be defined as the number n in the

Listing 7.8 different fixed point combinators for different orders of recurrences

```
#import std
#import nat
#import sol
#import tag

#fix general_type_fixer 0

ntre = ntre%WZnwAZ                    # a zero order recurrence

#fix general_type_fixer 1

xtre "s" = ("s",xtre "s")%drWZwlwAZ   # first order

#fix fix_lifter1 general_type_fixer 0

stre "s" = ("s",stre)%drWZwlwAZ       # zero order lifted by 1
```

minimum expression $(\ldots(t\ x_1)\ldots x_n)$ whereby the solution t yields an element of the semantic domain of interest.

All of these recurrences can be accommodated by the `#fix` directive, but an appropriate fixed point combinator must be supplied by the user, which depends in general on the order.

Calling conventions For an n-th order recurrence of the form

$$t\ =\ \texttt{"x1".}\ \ \ldots\ \ \texttt{"xn".}\ h(t,\texttt{"x1"},\ \ldots\ ,\texttt{"xn"})$$

or of the equivalent form

$$(\ldots(t\ \ \texttt{"x1"})\ldots\texttt{"xn"})\ =\ h(t,\texttt{"x1"},\ \ldots\ ,\texttt{"xn"})$$

or any combination, or for a recurrence that is semantically equivalent to one of these but expressed in a variable-free form, the argument to the fixed point combinator supplied by the user as a parameter to the `#fix` directive is the function

$$h'\ =\ \texttt{"t".}\ \ \texttt{"x1".}\ \ \ldots\ \ \texttt{"xn".}\ h(\texttt{"t"},\texttt{"x1"},\ \ldots\ ,\texttt{"xn"})$$

The fixed point combinator is required to return an argument y satisfying $y = h'(y)$.

Type expression recurrences Although a distinct fixed point combinator is required for every order, it may be possible to construct an ensemble of them from a single definition parameterized by a natural number, as a developer exploring these facilities will discover. Two ready made examples of semantic domains with complete hierarchies of fixed point combinators are functions and type expressions. For the sake of variety, the latter is illustrated in Listing 7.8.

The ensemble of fixed point combinators for type expressions is given by the function `general_type_fixer` defined in the `tag` library, which takes a number n to the n-th order fixed point combinator for type expressions. An example of a zero order recurrence is simply the recursive type expression for binary trees of natural numbers, `ntre`.

```
$ fun sol tag nxs.fun --m="1: (2: (),3: ())" --c ntre
1: (2: (),3: ())
```

A first order recurrence, `xtre`, defines the function that takes a type expression to a type of binary trees containing instances of the given type.

```
$ fun sol tag nxs.fun --m="1: (2: (),3: ())" --c "xtre %bL"
<true>: (<false,true>: (),<true,true>: ())
```

Because `xtre` is a function requiring a type expression as an argument, it is applied to the dummy variable in the recurrence. A similar function is implemented by `stre`.

```
$ fun sol tag nxs.fun --m="1: (2: (),3: ())" --c "stre %tL"
<&>: (<0,&>: (),<&,&>: ())
```

This recurrence is solved without recourse to higher order fixed point combinators, as explained below.

Lifting the order If a function p returning elements of a semantic domain P having a family of fixed point combinators F_n is the solution to a first order recurrence of the form

$$p \;=\; \texttt{"v".}\;\; h(p\;\; \texttt{"v"}, \texttt{"v"})$$

then one way to get it would be by evaluating

$$p \;=\; F_1\;\; \texttt{"f".}\;\; \texttt{"v".}\;\; h(\texttt{"f"}\;\; \texttt{"v"}, \texttt{"v"})$$

but another way would be

$$p \;=\; \texttt{"v".}\;\; F_0\;\; \texttt{"f".}\;\; h(\texttt{"f"}, \texttt{"v"})$$

because p occurs only by being applied to the dummy variable `"v"` in the recurrence. Most non-pathological recurrences satisfy this condition, and this transformation generalizes to higher orders.

The latter form may be advantageous because it depends only on the zero order fixed point combinator F_0, especially when higher orders are less efficient or unknown. All that's needed is to put the equation in the form

$$p \;=\; H\;\; \texttt{"f".}\;\; \texttt{"v".}\;\; h(\texttt{"f"}, \texttt{"v"})$$

so that it conforms to the calling conventions for the `#fix` directive (i.e., with H as the parameter), for some H depending only on F_0 and not higher orders of F.

This effect is achieved by taking $H = L_n\ F_m$, with a transformation L_n shifting n variables `"v"`, in this case 1.

$$L_1 = \texttt{"g". "h". "v". "g" "f". ("h" "f") "v"}$$

This transformation is valid for any fixed point combinator F_m and any order m. The family of transformations L_n is implemented by the `fix_lifter` function defined in the `sol` library distributed with the compiler, taking n as an argument.

Heterogeneous recurrences

Although this section begins with small contrived examples of functions and type expressions that could be expressed easily without recurrences, the difficulty of a manual solution quickly escalates in realistic situations involving mutual dependences among multiple declarations. It is compounded when the system involves multiple semantic domains and various orders of recurrences, to the point where a methodical approach may be needed.

In the most general case, each of m declarations can be associated with a separate fixed point combinator F_i for i ranging from 1 to m, in a source text organized as shown below.

$$\begin{array}{l} \texttt{\#fix}\ F_1 \\ x_1 \quad = \quad v_{11}\texttt{.} \quad \dots\ v_{1n}\texttt{.} \quad h_1(x_1 \dots x_m, v_{11} \dots v_{1n}) \\ \vdots \\ \texttt{\#fix}\ F_m \\ x_m \quad = \quad v_{m1}\texttt{.} \quad \dots\ v_{mn}\texttt{.} \quad h_m(x_1 \dots x_m, v_{m1} \dots v_{mn}) \end{array}$$

Although the declarations are shown here as lambda abstractions, any semantically equivalent form is acceptable, as noted previously.

- Each declared identifier x_i is defined by an expression $h_i(\dots)$ that may depend on itself and any or all of the other x's.
- Dummy variables v_{ij}, if any, are not shared among declarations, and their names need not be unique across them.
- There is no requirement for any solutions x_i to belong to the same semantic domain as any others, only that the corresponding fixed point combinator F_i is consistent with its type and the order of its declaration.
- A single `#fix` directive can apply to multiple declarations following it up to the next one.

In other respects, solving a system of recurrences automatically is no more difficult from the developer's point of view than solving a single one as in previous examples. In particular, there is no need for the developer to give any special consideration to heterogeneous or mutual recurrences when designing the fixed point combinator hierarchy for a particular semantic domain. It can be designed as if it were going to be used only to solve simple individual recurrences. Similar use may also be made of lifted fixed point combinators using the `fix_lifter` function.

7.6 Reflection

Most of the remaining compiler directives in Table 7.1 are hooks that can be made to perform any user defined operations not covered by the others. They come under the heading of reflection because they can access and inform the compiler's run-time data structures describing the application being compiled. Because this access permits unrestricted modifications, there is a possibility of disruption to the compiler's correct operation. Fortunately, safety is ensured by the user's capable judgment and intentions.

There is also a directive to interface with external development tools (e.g., "make" file generators and similar utilities) by providing a standardized access to user specified metadata.

7.6.1 The `#depend` directive

This directive takes any syntactically correct expression as a parameter, or at least an expression that can be parsed without causing an exception. The expression is never evaluated and is ignored during normal use. However, if the compiler is invoked with the `--depend` command line option, then the expression is written to standard output along with the source file name, and the rest of the file is ignored.

The reason this directive might be useful is that it allows any user defined metadata embedded in the source file to be extracted automatically by a shell script or other development tool without it having to lex the file.

For example, the directive can be used to list the names of the files on which a source file depends, so that a "make" utility can determine when it requires recompilation.

```
#import foo
#import bar

#depend foo bar
...
```

If a file `baz.fun` containing the above code fragment is compiled with the `--depend` command line option, the effect will be as follows.

```
$ fun baz.fun --depend
baz.fun:
foo bar
```

The script or development tool will need to parse this output, but that's easier than scanning the source file for `#import` directives. It's also more reliable if the directive is properly used because a file may depend on other files without importing them.

7.6.2 The `#preprocess` directive

This directive takes a function as a parameter that performs a parse tree transformation. The parse tree contains the declarations within the scope of the directive. When the tree is passed to the function during compilation, the function is required to return a tree of the same type.

The parse trees used by the compiler are of type `_token%T`, where the `token` record is defined in the `lag` library. For example, compilation of a file named `foobar.fun` containing the code fragment

```
#preprocess lag-_token%TM
x=y
```

would result in diagnostic message similar to the following.

```
fun:foobar.fun:1:1: ^: (
   token[
      lexeme: '#preprocess',
      filename: 'foobar.fun',
      filenumber: 3,
      location: (1,1),
      preprocessor: 399394%fOi&,
      semantics: 33568%fOi&],
   <
      ^: (
         token[
            lexeme: '=',
            filename: 'foobar.fun',
            filenumber: 3,
            location: (3,2),
            preprocessor: 4677323%fOi&,
            semantics: 13%fOi&],
         <
            ^:<> token[
               lexeme: 'x',
               filename: 'foobar.fun',
               filenumber: 3,
               location: (3,1),
               semantics: 12%fOi&],
            ^:<> token[
               lexeme: 'y',
               filename: 'foobar.fun',
               filenumber: 3,
               location: (3,3)]>)>)
```

Of course, in practice the function parameter to the `#preprocess` directive should do something more useful than dumping the parse tree as a diagnostic message. Effective use of this directive requires a knowledge of compiler internals as documented in Part IV of this manual. Possibly an even less useful example would be the following,

```
#preprocess *^0 &d.semantics:= ~&d.semantics|| 0!!!
```

which implements something like the infamous Fortran-style implicit declaration by giving every undeclared identifier used in any expression a default value of 0 rather than letting it cause a compile-time exception.

7.6.3 The `#postprocess` directive

This directive gives the user one last shot at any files generated by directives in its scope before they are written to external storage by the virtual machine. It is parameterized by a function that takes a list of files as input, and returns a list of files as a result. The files are represented as records in the form documented on page 260.

The following simple example will cause all output files in its scope to be written to the `/tmp` directory instead of being written relative to the current working directory or at absolute paths.

```
#postprocess * path:= ~path; ~&i&& :\<'tmp','');+ ~&h
```

This directive can be used intelligently without any further knowledge of compiler internals beyond the file record format documented in this chapter (unless of course it is used to modify the content of libraries or executable files significantly).

7.7 Command line options

An alternative way to use most of the directives documented in this chapter is by naming them on the command line when the compiler is invoked rather than by including them in the source text.

- An unparameterized directive like `#binary+` is expressed on the command line as `--binary` or `-binary`.
- A parameterized directive like `#cast` is written as `--cast "`t`"` on the command line for a parameter t, with quotes and escapes as required by the shell.

A directive given on the command line applies by default to every declaration in every source file as if it were inserted at the beginning of each. Unlike a directive in a file, there isn't the capability of switching it off selectively from the command line, even if applying it to every declaration is inappropriate, with two exceptions.

- Any directive selected on the command line can be made to apply to just one declaration by supplying an optional parameter stating the identifier of the declaration to which it applies. For example, `--cast` *foo*`,`*bar* specifies that the value of the identifier *bar* should be cast to the type *foo* and displayed as such.
- Some directives, such as `#cast` and `#show`, apply only to the last declaration within their scope in any case, so applying them to a whole file is the same as applying them only to the last declaration.

There are two other general differences between directives on the command line and directives in a file.

- Command line options other than `--trace` can be recognizably truncated, whereas directives in files must be spelled out in full.
- Command line options can also be ambiguously truncated if the ambiguity can be resolved by giving precedence to the options `--optimize`, `--show`, `--cast`, `--help`, `--archive`, `--parse`, and `--decompile`.

There are also some differences pertaining to specific directives.

- For the `--cast` command line option, the parameter is optional, but when used in a file as the `#cast` directive, the parameter is required.
- The `#hide` directives can be given only in a file and not on the command line.
- The `#depend` directive has a different effect from the `--depend` command line option, as noted in the Section 7.6.1.

Several other settings are selected only by command line options and not by directives in files. A complete list of command line options other than those corresponding to the directives documented previously is shown in Table 7.2. Those under the heading of customization allow normally fixed features of the language to be changed, such as the definitions of operators and type constructors. Effective use of these command line options requires a knowledge of the compiler internals, so their full discussion is deferred until Part IV. The remaining command line options in Table 7.2 are documented in the rest of this section.

7.7.1 Documentation

The two command line options `--version` and `--warranty` have the conventional effects of displaying short messages containing the compiler version number and non-warranty information. The `--help` option provides a variety of brief documentation interactively, and is intended as the first point of reference for real users.

The `--help` option by itself shows some general usage information and a list of all options with an indication of their parameters. It can also show more specific information when used with one of the following parameters. These parameters can be recognizably truncated.

- The `options` parameter shows a listing similar to table 7.2 that also includes the compiler directives accessible by the command line.
- The `directives` parameter shows a list of all compiler directives with short explanations.
- The `types` parameter shows a list of the mnemonics of all primitive types and type constructors with explanations (see Listing 4.10, page 175).

documentation		
`--help`	...	show information about options and features
`--version`		show the main compiler version number
`--warranty`		show a reminder about the lack of a warranty
verbosity		
`--alias`	...	use a specified command name in error messages
`--no-core-dumps`		suppress all core dump files
`--no-warnings`		suppress all warning messages
`--phase`	...	disgorge the compiler's run-time data structures
`--trace`		echo dialogs of the `interact` combinator
data display		
`--decompile`	...	suppress output files but display formatted virtual code
`--depend`		display data from `#depend` directives
`--parse`	...	parse and display code in fully parenthesized form
file handling		
`--archive`	...	compress binary output files and executables
`--data`	...	treat an input file as data instead of compiling it
`--gpl`	...	include GPL notification in executables and libraries
`--implicit-imports`		infer `#import` directives for command line libraries
`--main`	...	include the given declaration among those to be compiled
`--switches`	...	set application-specific compile-time switches
customization		
`--help-topics`	...	load interactive help topics from a file
`--pointers`	...	load pointer expression semantics from a file
`--precedence`	...	load operator precedence rules from a file
`--directives`	...	load directive semantics from a file
`--formulators`	...	load command line semantics from a file
`--operators`	...	load operator semantics from a file
`--types`	...	load type expression semantics from a file

Table 7.2: command line options; ellipses indicate an optional or mandatory parameter

- The usage `--help types,`t gives specific information about the type operator with the mnemonic t.
- The usages `--help types,`n, where n is `0`, `1`, or `2`, shows information only about primitive, unary, or binary type constructors, respectively.

- The `pointers` parameter lists the mnemonics for pointers and pseudo-pointers as documented in Chapter 2.
  - The usage `--help pointers,`p gives specific information about the pointer constructor with the mnemonic p.
  - The usages `--help pointers,`n, where n is `0`, `1`, `2`, or `3`, shows information only about pointers with those respective arities.

- Information about operators is displayed by the `--help` option with any of the parameters `prefix`, `postfix`, `infix`, `solo`, or `outfix`. The information is specific to the arity requested by the parameter.
  - Information about a specific known operator is requested by a usage such as `--help infix,"->"`.
  - If an operator contains the = character, the syntax is `--help=solo,"=="`.

- Information about operator suffixes for all operators of any arity is requested by `--help suffixes`. This parameter can also be used as above for information about a particular operator.

- A site-specific list of the virtual machine's libraries is requested by the `library` parameter, which shows a list of library names and function names (see Listing 1.10, page 46). This output is the same as that of `avram --e`.
  - A list of all functions in any library with a name beginning with the string *foo* is obtained by the usage `--help library,`*foo*.
  - A list of functions with names beginning with *bar* in libraries with names beginning with *foo* is obtained by `--help library,`*foo*`,`*bar*.

- The usage of `--help` s, where s is any string not matching any of those above, shows a listing of available options beginning with s, or shows the list of all options if there are none.

7.7.2 Verbosity

Several command line options can control the amount of diagnostic information reported by the compiler.

Warnings and core dumps

The `--no-warnings` and `--no-core-dumps` options have the obvious interpretations of suppressing warning messages and core dump files.

```
$ fun --main=0 --c %c
fun: writing `core'
warning: can't display as indicated type; core dumped
$ fun --main=0 --c %c --no-core-dumps
$ fun --main=0 --c %c --no-warnings
fun: writing `core'
```

Aliases

The `--alias` option changes the name of the application reported in diagnostic messages from `fun` to something else.

```
$ fun --m="~&h 0"
fun:command-line: invalid deconstruction
$ fun --alias serious --m="~&h 0"
serious:command-line: invalid deconstruction
```

This option is provided for the benefit of developers of application specific languages who want to use the compiler as a starting point and customize it.[2] The `alias` option would be hard coded into the shell script that invokes the compiler, so that end users need never suspect that they're using a functional programming language, even when something goes wrong. This effect can also be achieved simply by renaming the script.

Troubleshooting the compiler

The `--phase` option is of interest only to compiler developers. It takes a parameter of `0`, `1`, `2`, or `3`, and writes a binary file with the name `phase0` through `phase3`, respectively. The file contains a data structure of a self describing type (`%y`), expressing the program state at a particular phase of the operation. Normal compilation is not performed when this option is selected, but this operation may be time consuming due to the compression required for large data structures.

A useful technique to avoid including the `std` and `nat` libraries in the binary output file, thereby saving time and space, is to invoke the compiler by

`$ avram --par` ⟨*full path*⟩`/fun` ⟨*command line*⟩ `--phase` n

assuming the troublesome code in the source files in the command line has been narrowed down enough not to depend on the standard libraries.

[2]or simplify it for a user base they consider less clever than themselves

Debugging client/server interactions

The `--trace` option is passed through to the virtual machine, requesting all characters exchanged between an application using the `interact` combinator and an external command line interpreter to be displayed on the console along with some verbose diagnostic information. Unlike most command line options, `--trace` must be written out in full and may not be truncated. This option is useful mainly for debugging. See the `avram` reference manual for further information. Here is an example using a function from the `cli` library.

```
$ fun cli --m=now0 --c --trace
opening bash
waiting for 36 32
⋮
-> $ 36
->   32
matched
<- e 101
<- x 120
<- i 105
<- t 116
<-   10
waiting for nothing
matched
closing bash
'Tue, 19 Jun 2007 23:44:30 +0100'
```

7.7.3 Data display

A small selection of command line options can be used to display information specific to a given program source text or expression. The `--cast` command line option, seen in many previous examples, is derived from the `#cast` directive documented in Section 7.4.1, hence not repeated here. The same goes for the `--show` option, which is also frequently used (Section 7.4.2). The others are summarized below.

- The `--decompile` option shows the virtual machine code for the last expression compiled, assuming it is a function. The expression can come from either the source text or from a `--main` option. The code is expressed using the mnemonics from the `cor` library, (Listing 3.1, page 113) and documented extensively in the `avram` reference manual. This option is similar to `--cast %f`, except that it displays the full declaration.
- The `--depend` option displays the expression used as a parameter to any `#depend` directives in the source texts on standard output, prefaced by the name of the source file. See Section 7.6.1 for more information and motivation.

- The `--parse` option causes an expression to be displayed in fully parenthesized form, thereby settling questions of operator precedence and associativity. (See page 179 for motivation.) The expression is not evaluated and may contain undefined identifiers.
  - If a parameter is supplied with the `--parse` option, as in `--parse x`, then the expression declared with the identifier of the parameter `x` is parsed.
  - If the optional parameter is the literal character string "`all`", then every declaration in every source file is parsed and displayed.
  - If a `--main` option is used at the same time as a `--parse` option with no parameter, then expression in the `--main` parameter is parsed.
  - If no `--main` option is present, and the `--parse` option has no parameter, the last declaration in the last file is parsed.

7.7.4 File handling

The remaining command line options in Table 7.2 pertain to the handling of input and output files.

Output files

The `--archive` and `--gpl` options are specific to library files and executables (i.e., those generated by the `#library` or `#executable` directives). Each takes an optional numerical parameter.

`--archive` This option causes a library file to be compressed, or an executable code file to be stored in a compressed self-extracting form. The optional parameter is the granularity of compression, which has the same interpretation as the granularity of compressed types explained on page 169. The default behavior without a parameter is maximum compression, which is usually the best choice. Compression is usually a matter of necessity for any non-trivial application, without which the file size explodes, and the memory requirements even more so.

- Compressed libraries are indistinguishable from uncompressed libraries when imported by the `#import` directive or dereferenced with the dash operator.
- Compressed executables are indistinguishable from uncompressed executables, because they are automatically made self-extracting. There may be a small run-time overhead incurred by the extraction when the application is launched.

`--gpl` This option causes a notification to be inserted into the preamble of every library or executable file generated in the course of a compilation to the effect that its distribution terms are given by the General Public License as published by the Free Software Foundation. The optional parameter is the version number of the license, with versions 2 and 3

character	spelling
`0`	`zero`
`1`	`one`
`2`	`two`
`3`	`three`
`4`	`four`
`5`	`five`
`6`	`six`
`7`	`seven`
`8`	`eight`
`9`	`nine`
`(`	`paren`
`)`	`thesis`
`.`	`dot`
`,`	`comma`
`-`	`dash`
`;`	`semi`
`@`	`at`
`%`	`percent`
	`space`

Table 7.3: rewrite rules for special characters in file names

being the only valid choices at this writing. The default is version 3. Only the specified version is applicable, as the text does not include the provision for "any later version".

Needless to say, this option is optional. It should not be selected unless the author intends to distribute the software on these terms. One alternative is to keep it only for personal use. Another is to distribute it subject to a non-free license. In the latter case, the software must not depend on any code from the standard libraries distributed with the compiler, which would ordinarily be copied into it as a consequence of compilation. The specifications in Part III of this manual will enable a clean-room re-implementation of these libraries for proprietary redistribution if necessary.

Input files

When the compiler is invoked with multiple input files, the default behavior is to treat the binary files as data and to compile the text files as source code. For this purpose, binary files are those that conform to the format used in files generated by the directives `#library`, `#binary`, and `#executable`, and text files are any other files, even if they contain unprintable characters.

No explicit i/o operations are required in the source files to access the contents of the data files. Instead, the contents of the data files are accessible in the source files as the values of pre-declared identifiers derived from the file names.

- If a data file name contains only alphabetic characters, the identifier associated with it is the file name.

- If the name of a data file contains any characters that are not valid in identifiers, these characters are rewritten according to Table 7.3.
- The rewritten character are bracketed by underscores in the identifier. For example, a data file named `foo.bar` would be accessed as the identifier `foo_dot_bar`.
- The default file suffix for library files, `.avm`, is ignored, so that identifiers ending with `_dot_avm` are not needed.

The remaining command line options in Table 7.2 affect the way input files are treated.

`--data` This option can be used to override the default behavior for text files by causing them to be treated as data files instead of being compiled. The value of the identifier associated with a text file will be a list of character strings storing the contents of the file.

The `--data` option is unusual in that its placement on the command line is significant. It must immediately precede the name of the file that is to be treated as data. It pertains only to that file and not to any files given subsequently on the command line. If there are multiple text files to be treated as data files, each one must be preceded by a separate `--data` option.

`--implicit-imports` When this option is selected, all files with suffixes of `.avm` on the command line are detected. These files are required to be valid library files generated by the `#library` directive during a previous compilation. An `#import` directive is constructed with the name of each library file, and this sequence of `#import` directives is inserted at the beginning of each source file. The resulting effect is that the code in the source files may refer to symbols within the library files as if they were locally declared, without having to import them.

`--switches` This option takes a comma separated sequences of parameters, and causes the predeclared identifier `__switches` to evaluate to them in any source text being compiled, as this example shows.

```
$ fun --m=__switches --switches=foo,bar,baz --c
<'foo','bar','baz'>
```

The type of the predeclared identifier `__switches` is always a list of character strings. See page 249 for more information and motivation.

`--main` This option is used in many previous examples. Its purpose is to allow for easy interactive compilation of short expressions directly from the command line without requiring them to be stored in a file.

- The parameter to the `--main` option contains the text be compiled, which can be either a single expression or a sequence of one or more declarations.

- In the case of a single expression, x, the text of the parameter is compiled as if it contained the declaration `main =` x.
- The language syntax is the same for `--main` expressions as for ordinary source text, but it may need to be quoted or escaped to prevent interpretation by the shell.
- The `--main` expression may use identifiers declared in any libraries mentioned on the command line, as well as the `std` and `nat` libraries, without need of an `#import` directive.
- The `--main` expression may use identifiers declared in the last source file named on the command line, if any, without need of an `#export` directive.

7.8 Remarks

This chapter concludes Part II of this manual on Language Elements. These specifications are expected to remain fairly stable for the forseeable future, with most new development work concentrating on the standard libraries documented in Part III.

Readers with a good grasp of this material are well posed to begin developing practical applications with Ursala. Please use your powers wisely and only for the benefit of all mankind.

Part III

Standard Libraries

I require the exclusive use of this room, as well as that drafty sewer you call the library.

Sheridan Whiteside, *The man who came to dinner*

8
A general purpose library

Most applications in this language as in others are not developed *ab initio* but from a reusable code base of tried and tested components. A growing collection of library modules packaged and maintained along with the compiler provides a variety of helpful utilities in the way of functions, combining forms, and data structure specifications.

8.1 Overview of packaged libraries

There are three subdirectories in the main distribution package populated with `.avm` virtual code library files, these being the `src/`, `lib/`, and `contrib/` directories.

- The `contrib/` directory contains libraries for experimental, illustrative, or archival purposes, that are not necessarily maintained and are not documented in this manual.
- The `src/` directory contains libraries necessary to bootstrap the compiler. They are maintained but are unlikely to be of any independent interest except for the `std` and `nat` libraries. Some *ad hoc* documentation about them suitable for compiler developers is provided in Part IV.
- The `lib/` directory contains the libraries that are considered important complements to the core functionality of the language. These are maintained and meticulously documented in this chapter and the succeeding ones in Part III.

8.1.1 Installation assumptions

In the recommended installation, all `.avm` files in `src/` and `lib/` are stored in the host filesystem under `/usr/lib/avm/` or `/usr/local/lib/avm/`, where they are automatically detected by the virtual machine with no path specification required.

- These files are architecture independent and therefore could be exported on a network filesystem for use by multiple clients without binary code compatibility issues.
- Non-standard installations may require the the user or system administrator make arrangements for specifying the library file paths when invoking the compiler. See Section 1.3.1 on page 51 for a related discussion.

8.1.2 Documentation conventions

Each library is documented in a separate chapter, even though some chapters may be very short. The style is that of a reference manual, often with little more than a catalog of descriptions of the library functions and data structures. The emphasis is more on accuracy and completeness than motivation or literary merit, and this style is most conducive to maintaining current information about an evolving code base. These chapters need not be read sequentially, but they take a working knowledge of the material in Part II for granted.

The `std` and `nat` libraries are under the `src/` directory in the packaged distribution because they are necessary for bootstrapping the compiler, but they are also suitable for more general use so they are documented in Part III.

The remainder of this chapter documents the `std` library. Unlike most other libraries, this one can be imported into any source text without being given as a command line parameter to the compiler, because it is automatically supplied by the shell script that invokes the compiler.

8.2 Constants

The standard library defines three constants that are useful for input parsing and validation.

`characters`

the list of 256 characters (type `%c`) ordered by their ISO codes

`letters`

the list of 52 upper and lower case alphabetic characters, `a`...`z``A`...`Z`, with the lower case characters first

`digits`

the list of ten decimal digits `0`...`9`

A predicate that tests whether its argument is a digit could be coded as `-=digits`, as an example.

Other constants, such as `true` and `false`, are also defined by the standard library, because all symbols in the `cor` library (Listing 3.1, page 113) are included in it.

8.3 Enumeration

Two functions tangentially related to the idea of enumeration are the following.

upto

Given a natural number n, this function returns a list containing every possible datum of any type whose binary representation size measured in quits doesn't exceed n

For example, there are 9 data with a size up to three.

```
$ fun --m=upto3 --c %tL
<
   0,
   &,
   (0,&),
   (&,0),
   (0,(0,&)),
   (0,(&,0)),
   (&,&),
   ((0,&),0),
   ((&,0),0)>
```

This function is useful for exhaustively testing code that operates on small data structures or pointers. However, it should be used with caution because the number of results increases exponentially with the size n, being given by $\sum_{i=0}^{n} f(i)$, where $f(0) = 1$ and

$$f(i) = \sum_{j=0}^{i-1} f(j)f(i-j)$$

for $i > 0$.

enum

This function takes a set of data and returns a type expression for the type whose instances are the data. See page 173 for an example.

8.4 File Handling

Executable applications that have a command line interface or that generate output files are expressed as functions that observe consistent calling conventions. The standard library provides a small set of data structure declarations and functions in support of these conventions.

8.4.1 Data Structures

The following four identifiers are record mnemonics. Their usage is explained with examples starting on page 258, but they are briefly recounted here for reference.

invocation

A record of this form passed to any command line application generated by the `#executable` directive with a parameterized interface. The record consists of two fields, `command` and `environs`. The latter contains a module of character strings specifying the environment variables.

command_line

A record of this form makes up the `command` field of an invocation record. It has two fields, `files` and `options`.

file

A list of records of this form is stored in the `files` field in a `command_line` record. It has four fields describing a file, which are called `stamp`, `path`, `preamble` and `contents`. The interpretation of these fields is explained on Page 260.

option

A list of these records is stored in the `options` field of a `command_line` record. Its four fields are called `position`, `longform`, `keyword`, and `parameters`. Their interpretations are explained on page 261.

8.4.2 Functions

Two further functions are intended to facilitate generating output files or other possible uses.

gpl

This function takes a version number as a character string (usually `'2'` or `'3'`), and returns a list of character strings containing the standard General Public License notification for the corresponding version, "This program is free software ...". If an empty string is supplied as an argument, the version number defaults to 3.

dot

This function is meant to be used in an output file generating directive of the form `#output dot`⟨*suffix*⟩ ⟨*function*⟩ as explained on page 265.

8.5 Control Structures

A small group of control structures comparable to those in other languages is specified by the combining forms documented in this section. These are not built into the language but defined as library functions.

8.5.1 Conditional

An idea originated by Tony Hoare, case statements are useful as a structured form of nested conditionals whose predicates test the argument against a constant. (This construct is more restrictive than the cumulative conditional combinator, which allows general predicates as explained on page 191.) In typical usage, a function H of the form

$$
\begin{array}{lll}
H & = & \texttt{(case } f\texttt{) (} \\
 & & \quad \texttt{<} \\
 & & \qquad k_0\texttt{:}\ g_0\texttt{,} \\
 & & \qquad \vdots \\
 & & \qquad k_n\texttt{:}\ g_n\texttt{>,} \\
 & & \quad h\texttt{)}
\end{array}
$$

applied to an argument x first computes the value $k = f(x)$, and then tests k against each possible k_i in sequence. For the first matching k_i, the corresponding function $g_i(x)$ is evaluated and its result is returned. If no match is found, $h(x)$ is returned. Note that g_i or h is applied to the original argument, x, not to k, which is only an intermediate result that is not returned. Evaluation is non-strict insofar as only the g_i for the matching k_i is evaluated, if any, and h is not evaluated unless no match is found.

Two forms of `case` statement defined in the standard library differ in the nature of the test, and the third generalizes both of these.

`case`

This function takes a function f as an argument and returns a function that maps a pair $\texttt{(<}k_0\texttt{:}\ g_0\texttt{,}\ \dots\ k_n\texttt{:}\ g_n\texttt{>,}h\texttt{)}$ to a function H as above. In terms of the foregoing notation, a match between k and k_i occurs precisely when they are equal in the sense described on page 78.

`cases`

This function follows the same calling convention as the `case` function, above, but differs in the semantics of the resulting H. In order for a match to occur between the temporary value k and a constant k_i, the constant k_i must be a list or a set of which k is a member.

A short example of the `cases` function is the following, which takes a character or anything else as an argument and returns a string describing its classification, if recognized.

```
classifier = cases~&\'unrecognized'! <
   'aeiouAEIOU': 'vowel'!,
   letters: 'consonant'!,
   digits: 'digit'!>
```

Note that because the order in which the cases are listed is significant, the patterns may overlap without ambiguity. If the patterns are mutually disjoint, use of braces is preferable to angle brackets as a matter of style and clarity.

The concept of a case statement generalizes to arbitrary matching criteria beyond equality and membership.

gcase

Given a any function p computing a predicate, this function returns a case statement constructor in which a match between k and k_i is deemed to occur when $p(k, k_i)$ holds, where k and k_i are as in the preceding explanations.

For example, the first `case` function can be defined as `gcase ==`, and the second one, `cases`, can be defined as `gcase -=`. A case statement based membership in numerical intervals would be another obvious example.

lesser

This function takes a binary relational predicate to the corresponding binary minimization function. For any funciton p, the function `lesser` p takes an argument (x, y) to x if $p(x, y)$ is non-empty, and to y otherwise.

8.5.2 Unconditional

Most of the basic functional combining forms in the language are provided by the operators documented in Chapter 6, but several are expressible as follows.

gang

This function takes a list of functions to a function returning a list. The function $\texttt{gang<}f_0\texttt{, ... ,}f_n\texttt{>}$ applied to an argument x returns the list. $\texttt{<}f_0\ x\texttt{, ... ,}f_n\ x\texttt{>}$ This function is equivalent to $\texttt{<.}f_0\texttt{, ... ,}f_n\texttt{>}$. (See page 194 for an example.)

associate_left

This function takes any function operating on a pair to a function that operates on a list. The function `associate_left` f returns `<>` for an empty list and returns the head of list with only one item. For lists with more than one item, it satisfies the recurrence

$$(\texttt{associate\_left}\ f)\ a:b:x = (\texttt{associate\_left}\ f)\ (f(a,b)):x$$

A simple example of this function would be

```
$ fun --m="associate_left~& 'abcdef'" --c
((((('a,'b),'c),'d),'e),'f)
```

fused

The argument to this function should be a record initializing function r (i.e., something declared with the `::` operator as explained in Section 4.2). The result is a function that takes a pair of records (x, y) each of type $\_r$ and returns a record z also of type $\_r$. The result z consists of the non-empty fields from x and the remaining fields, if any, from y, followed with initialization by the function r.

A short example of this function is as follows.

```
$ fun --m="r::a %n b %n x=fused(r)/r[a: 1] r[b: 2]" --c _r
r[a: 1,b: 2]
```

8.5.3 Iterative

A couple of functions useful mainly for debugging can be used to iterate a function a fixed number of times.

rep

This function takes a natural number n as an argument, and returns a function that maps a given function f to the composition of f with itself n times (or equivalent). If $n = 0$, the result of $(\texttt{rep}\ \ n)\ \ f$ is the identity function.

The following example demonstrates the `rep` function by inserting a zero at the head of a list five times.

```
$ fun --m="rep5~&NiC <1>" --c %nL
<0,0,0,0,0,1>
```

next

This function takes a natural number n and returns a function that takes a given function f to the equivalent of `<.rep0` f`, ... ,rep`$(n-1)$ f`>`. That is, the result of $(\texttt{next}\ \ n)\ \ f$ is a function returning a list of length n whose i-th item is the result of i iterations of f on the argument, starting from zero.

An example of the `next` function following on from the previous example is as shown.

```
$ fun --m="next5~&NiC <1>" --c %nLL
<<1>,<0,1>,<0,0,1>,<0,0,0,1>,<0,0,0,0,1>>
```

8.5.4 Random

Three functions are defined in the standard library for generating pseudo-random data according to some specified distribution. The underlying random number generator is the Mersenne Twister algorithm provided by the virtual machine's `mtwist` library, as documented in the `avram` reference manual.

arc

This function, mnemonic for "arbitrary constant", takes any set as an argument, and constructs a program that ignores its input but returns a pseudo-randomly chosen member of the set. The value returned by the program may be different for each execution, with all members of the set being equally probable.

An example of the `arc` function is given by the following expression.

```
$ fun --m="arc<0,1,2>* '--------'" --c
<0,2,1,1,0,1,2,1>
```

choice

This function takes a set of functions as an argument and constructs a program that chooses one to apply to its input each time it is invoked. A simulated non-deterministic choice is made, with all choices being equally probable.

This example shows a choice of three functions applied to a string, with a different choice made for each execution.

```
$ fun --m="choice{~&,~&x,~&iiT} 'foo'" --c %s
'foofoo'
$ fun --m="choice{~&,~&x,~&iiT} 'foo'" --c %s
'foo'
$ fun --m="choice{~&,~&x,~&iiT} 'foo'" --c %s
'oof'
```

stochasm

This function takes a set $\{p_0\colon f_0\ \dots p_n\colon f_n\}$ of assignments of probabilities to functions, and constructs a program that simulates a non-deterministic choice among the functions each time it is invoked. Preference is given to each function in proportion to its probability. Probabilities p_i needn't sum to unity but they must be non-negative. They may be either floating point or natural numbers (type `%e` or `%n`).

Two examples of the `stochasm` function demonstrate filters that lose twenty and seventy percent of their input on average.

```
$ fun --m="stochasm{0.8: ~&iNC,0.2: ''!}*= letters" --c
```

```
'abcdhijkmopqrsvwxzADEGHIJKLMNOPQRSTVXZ'
$ fun --m="stochasm{0.3: ~&iNC,0.7: ''!}*= letters" --c
'dehilnosDFLMNOSVY'
```

8.6 List rearrangement

A collection of functions defined in the standard library for operating on lists supplements the operators and pseudo-pointers in the core language.

8.6.1 Binary functions

These functions take a pair of lists to a list.

zip

> Given a pair of list $(\langle x_0 \dots x_n\rangle, \langle y_0 \dots y_n\rangle)$ of the same length, this function returns the list of pairs $\langle (x_0, y_0) \dots (x_n, y_n)\rangle$. If the lists are of unequal lengths, the function raises an exception with the diagnostic message "`bad zip`".

The `zip` function is equivalent to the `~&p` pseudo-pointer (page 75).

zipt

> This function performs a truncating zip operation. It follows a similar calling convention to the `zip` function, above, but does not require the lists to be of equal length. If the lengths are unequal, the shorter list is zipped to a prefix of the longer one.

The `zipt` function is equivalent to the one used in an example on Page 73.

gcp

> This function returns the greatest common prefix of a pair of lists, which is the longest list that is a prefix of both of them.

An example of an application of the `gcp` function is the following.

```
$ fun --m="gcp/'abc' 'abd'" --c %s
'ab'
```

8.6.2 Numerical

The function in this section perform operations on lists that are parameterized by natural numbers.

`iol`

Given any list, this function returns a list of consecutive natural numbers starting with zero that has the same length as its argument.

This function is exemplified in the following expression.

```
$ fun --m="iol 'catabolic'" --c
<0,1,2,3,4,5,6,7,8>
```

`num`

This function takes any list as an argument and returns a list of pairs in which the left sides form a consecutive sequence of natural numbers starting from zero, and the right sides are the items of the argument in their original order. It is equivalent to the function `^p/iol ~&`.

The `num` function numbers the items of a given list as shown.

```
$ fun --m="num 'abcde'" --c %ncXL
<(0,`a),(1,`b),(2,`c),(3,`d),(4,`e)>
```

`skip`

Given a pair (n, x), where n is a natural number and x is a list, this function returns a copy of the list x with the first n items deleted. If x does not have more than n items, the empty list is returned.

`take`

Given a pair (n, x), where n is natural number and x is a list, this function returns a copy of the list x with all but the first n items deleted. If x does not have more than n items, the whole list is returned.

`block`

Given a number n, this function returns a function that maps any list x into a list of lists y such that `~&L` $y = x$, and every item of y has a length of n except possibly the last, which may have a length less than n.

An example of the `block` function is the following.

```
$ fun --m="block3 'abcdefghijkl'" --c %sL
<'abc','def','ghi','jkl'>
```

swin

> Given a number n, this function returns a function that maps any list x into a list of lists y whose i-th item is the length n substring of x beginning at position i.

The function name is mnemonic for "sliding window". An example of the `swin` function is the following.

```
$ fun --m="swin3 'abcdef'" --c %sL
<'abc','bcd','cde','def'>
```

8.6.3 General

Some further list editing operations parameterized by functions or constants are documented in this section. These include functions for padded zips, variations on flattening and unflattening, sorting, and conditional truncation.

zipp

> This function takes a constant k to a function that zips two lists together of arbitrary length by padding the shorter one with copies of k if necessary. It satisfies the following recurrences.

$$\begin{aligned}(\texttt{zipp}\ k)\ (\texttt{<>},\texttt{<>}) &= \texttt{<>}\\ (\texttt{zipp}\ k)\ (a:x,\texttt{<>}) &= (a,k):((\texttt{zipp}\ k)\ (x,\texttt{<>}))\\ (\texttt{zipp}\ k)\ (\texttt{<>},b:y) &= (k,b):((\texttt{zipp}\ k)\ (\texttt{<>},y))\\ (\texttt{zipp}\ k)\ (a:x,b:y) &= (a,b):((\texttt{zipp}\ k)\ (x,y))\end{aligned}$$

This example shows the `zipp` function zipping two lists of natural numbers by padding the shorter one with zeros.

```
$ fun --m="zipp0/<1,2,3> <4,5,6,7,8>" --c %nWL
<(1,4),(2,5),(3,6),(0,7),(0,8)>
```

pad

> This function takes a constant k to a function that takes a list of lists of differing lengths to a list of lists of the same length by appending copies of k to those that are shorter than the maximum. It is defined as follows.

```
pad "k" = ~&i&& ~&rSS+ zipp"k"^*D\~& leql$^
```

This example shows how a list of lists of lengths 2, 1, and 3 is transformed to a list of three lists of length three by padding the shorter lists.

```
$ fun --m="pad1 <<0,1>,<2>,<3,4,5>>" --c %nLL
<<0,1,1>,<2,1,1>,<3,4,5>>
```

mat

This function takes a constant k of type t to a function that flattens a list of type t`%LL` to a list of type t`%L` after inserting a copy of <k> between consecutive items. It can be defined as `:-0+ ^|T/~&+ //:`, among other ways.

The following example shows how a ten is inserted after every three numbers in the list of natural numbers from 0 to 9.

```
$ fun --m="mat10 block3 <0,1,2,3,4,5,6,7,8,9>" --c %nL
<0,1,2,10,3,4,5,10,6,7,8,10,9>
```

sep

This function serves as something like an inverse to the `mat` function, in that (`mat` k)`+ sep` k is equivalent to the identity function. For a given separator k, the function `sep` k scans a list for occurrences of k, and returns the list of lists of intervening items.

The `sep` function can be used in text processing applications to implement a simple lexical analyzer. In this example, a path name containing forward slashes is separated into its component directory names.

```
$ fun --m="sep\`/ 'usr/share/doc/texlive-common'" --c %sL
<'usr','share','doc','texlive-common'>
```

Note that the backslash is there to suppress interpretation of the backquote character by the shell, and would not be used if this code fragment were in a source file.

psort

This function, mnemonic for "priority sort", takes a list of relational predicates <$p_0 \dots p_n$> to a function that sorts a list x by the members of p in order of decreasing priority. That is, the ordering of any two items of x is determined by the first p_i whereby they are not mutually related.

The `psort` function is useful for things like sorting a list of time stamps by the year, sorting the times within each year by the month, sorting the times within each month by the day, and so on. This example shows how a list of strings is lexically sorted with higher priority to the second character.

```
$ fun --m="psort<lleq+~&bth,lleq+~&bh> <'za','ab','aa'>" -c
<'aa','za','ab'>
```

The lexical order relational predicate `lleq` is documented subsequently in this chapter.

rlc

This function, mnemonic for “run length code”, takes a relational predicate as an argument and returns a function that separates a list into sublists. The predicate is applied to every pair of consecutive items, and any two related items are classed in the same sublist. The cumulative concatenation of the sublists recovers the original list.

An example of the `rlc` function that collects runs of identical list items is the following.

```
$ fun --m="rlc~&E <0,0,1,0,1,1,1,0,1,0,0>" --c %nLL
<<0,0>,<1>,<0>,<1,1,1>,<0>,<1>,<0,0>>
```

This function could be carried a step further to compute the conventional run length encoding of a sequence by `^(length,~&h)*+ rlc~&E`, which would return a list of pairs with the length of each run on the left and its content on the right.

takewhile

This function takes a predicate as an argument, and returns a function that truncates a list starting from the first item to falsify the predicate.

In this example, the remainder of a list following the first run of odd numbers is deleted.

```
$ fun --m="takewhile~&h <1,3,5,2,4,7,9>" --c %nL
<1,3,5>
```

skipwhile

This function takes a predicate as an argument, and returns a function that deletes the maximum prefix of a list whose items all falsify the predicate.

In this example, the odd numbers at the beginning of a list are deleted.

```
$ fun --m="skipwhile~&h <1,3,5,2,4,7,9>" --c %nL
<2,4,7,9>
```

Recall that `~&h` tests the least significant bit of the binary representation of a natural number.

8.6.4 Combinatorics

Various functions relevant to combinatorial problems are defined in the standard library. These include functions for computing transitive closures and cross products, permutations, combinations, and powersets.

closure

> Given a relation represented as a set of pairs, this function computes the transitive closure of the relation. The transitive closure of a relation R is defined as the minimum relation containing R for which membership of any (x, y) and (y, z) implies membership of (x, z).

A simple example of the `closure` function is the following.

```
$ fun --m="closure{('x','y'),('y','z')}" --c %sWS
{('x','y'),('x','z'),('y','z')}
```

cross

> This function takes a pair of sets to their cartesian product. The cartesian product of a pair of sets (S, T) is defined as the set of all pairs (x, y) for which $x \in S$ and $y \in T$. This function is equivalent to the `~&K0` pseudo-pointer (page 91).

permutations

> Given a list x of length n, this function returns a list of lists containing all possible orderings of the members in x. The result will have a length of $n!$ (that is, $1 \cdot 2 \cdot \cdots \cdot n$), and will contain repetitions if x does.

An example of the `permutations` function for a three item list is the following.

```
$ fun --m="permutations 'abc'" --c %sL
<'abc','bac','bca','acb','cab','cba'>
```

powerset

> This function takes any set to the set of all of its subsets. The cardinality of the powerset of a set of n elements is necessarily 2^n.

This example shows the powerset of a set of three natural numbers.

```
$ fun --m="powerset {0,1,2}" --c %nSS
{{},{0},{0,2},{0,2,1},{0,1},{2},{2,1},{1}}
```

choices

> Given a pair (s, k), where s is a set and k is a natural number, this function returns the set of all subsets of s having cardinality k. For a set s of cardinality n, the number of subsets will be

$$\left(\begin{array}{c} n \\ k \end{array} \right) = \frac{n!}{k!(n-k)!}$$

For a very small example, the set of all three element subsets from a universe of cardinality 4 is illustrated as shown.

```
$ fun --m="choices/'abcd' 3" --c %sL
<'abc','abd','acd','bcd'>
```

cuts

Given a pair (s, k), where s is a list and k is a natural number, this function finds every possible way of separating s into $k+1$ non-empty consecutive parts. Each alternative is encoded as a list of sublists whose concatenation yields s. A list containing all such encodings is returned.

This example shows all possible subdivisions of a nine item lists into three consecutive parts.

```
$ fun --m="cuts('abcdefghi',2)" --c %sLL
<
   <'a','b','cdefghi'>,
   <'a','bc','defghi'>,
   <'a','bcd','efghi'>,
   <'a','bcde','fghi'>,
   <'a','bcdef','ghi'>,
   <'a','bcdefg','hi'>,
   <'a','bcdefgh','i'>,
   <'ab','c','defghi'>,
   <'ab','cd','efghi'>,
   <'ab','cde','fghi'>,
   <'ab','cdef','ghi'>,
   <'ab','cdefg','hi'>,
   <'ab','cdefgh','i'>,
   <'abc','d','efghi'>,
   <'abc','de','fghi'>,
   <'abc','def','ghi'>,
   <'abc','defg','hi'>,
   <'abc','defgh','i'>,
   <'abcd','e','fghi'>,
   <'abcd','ef','ghi'>,
   <'abcd','efg','hi'>,
   <'abcd','efgh','i'>,
   <'abcde','f','ghi'>,
   <'abcde','fg','hi'>,
   <'abcde','fgh','i'>,
   <'abcdef','g','hi'>,
```

```
   <'abcdef','gh','i'>,
   <'abcdefg','h','i'>>
```

The result is ordered by length of the first sublists with different lengths.

words

This function takes a natural number n to a function that takes an alphabet a to an enumeration of all length n sequences of members of a.

The `words` function differs from the `choices` function described previously insofar as order is significant and repetitions are allowed. Hence, an expression of the form `words(n) a` will evaluate to a list of length $|a|^n$, where $|a|$ is the cardinality of a. Here is an example usage.

```
$ fun --m="words5 '01'" --c
<
   '00000',
   '00001',
   '00010',
   '00011',
   '00100',
   '00101',
   '00110',
   '00111',
   '01000',
   '01001',
   '01010',
   '01011',
   '01100',
   '01101',
   '01110',
   '01111',
   '10000',
   '10001',
   '10010',
   '10011',
   '10100',
   '10101',
   '10110',
   '10111',
   '11000',
   '11001',
   '11010',
   '11011',
```

```
'11100',
'11101',
'11110',
'11111'>
```

8.7 Predicates

Various primitive functions and combinators are defined in the standard library to assist in applications needing to compute truth values or decision procedures.

8.7.1 Primitive

A number of predicates that are mostly binary relations are provided by the definitions documented in this section.

- As a matter of convention, predicates may return any non-empty value when said to hold or to be true, and will return the empty value `()` when false.
- These predicates are false in all cases where the descriptions do not stipulate that they are true.
- Equality is in the sense described on page 78.
- Read "if" as "if and only if".

`eql`

This predicate holds for any pair of lists (x, y) in which x has the same number of items as y, counting repeated items as distinct.

`leql`

This predicate holds for any pair of lists (x, y) in which x has no more items than y, counting repeated items as distinct.

`intersecting`

This predicate is true of any pair of lists or sets (x, y) for which there exists an item that is a member of both x and y. It is logically equivalent to the `~&c` pseudo-pointer but faster (page 77).

`subset`

This predicate is true of pairs of sets or lists (s, t) wherein every element of s is also an element of t. If s is empty, then it is vacuously satisfied.

`substring`

This predicate is true of any pair of lists (s,t) for which there exist lists x and y such that $x{-}{-}s{-}{-}y$ is equal to t.

`suffix`

This predicate is true of any pair of strings or lists (s,t) for which there exists a list x such that $x{-}{-}s$ is equal to t.

`lleq`

This function computes the lexical partial order relation on characters, strings, lists of strings, and so on. Given a pair of strings (s,t), the predicate is true if s alphabetically precedes t. For a pair of characters (s,t), the predicate holds if the ISO code of s is not greater than that of t.

`indexable`

This predicate is true of any pair (p,x) for which ~p x can be evaluated without causing an exception. This relationship is best understood by envisioning both x and p as transparent types and considering it recursively.

- If p is a pair that is non-empty on both sides, then it is indexable with x only if both sides are individually indexable with it.
- If p is empty on one side and not the other, then it is indexable with x only if the non-empty side is indexable with the corresponding side of x.
- If p is empty on both sides, then it is always indexable with x.

`singly_branched`

This predicate is true of the empty pair `()`, and of any pair that is empty on one side and singly branched on the other.

8.7.2 Boolean combinators

The boolean operations are most conveniently obtained by combinators taking predicates to predicates rather than by first order functions. Predicates used as arguments to the functions in this section could be any of those documented in the previous section, as well as any user defined predicates.

Each of these predicate combinators is unary in the sense that it takes a single predicate as an argument and returns a single predicate as a result. However, the predicate it returns may operate on a pair of values. In that case, evaluation is non-strict in that only the left value is considered where it suffices to determine the result.

Similar conventions to those of the previous section regarding truth values apply here as well.

`not`

Given a predicate p, this function constructs a predicate that is true whenever p is false, and vice versa.

`both`

Given a predicate p, this function constructs a predicate that applies p to both sides of a pair, and is true only if the result is true in both cases.

`neither`

Given a predicate p, this function constructs a predicate that applies p to both sides of a pair, and returns a true value if the result of both applications is false.

`either`

Given a predicate p, this function constructs a predicate that applies p to both sides of a pair, and returns a true value if the result of at least one application is true.

8.7.3 Predicates on lists

These combinators take an arbitrary predicate as an argument and return a predicate that operates on a list.

`ordered`

Given a relational predicate p, this function constructs a predicate that is true if its argument is a list whose items form a non-descending sequence with respect to p. That is, (`ordered` p) x is true if x is equal to $p\!-\!<\ x$. If p is a partial order relation, then `ordered` p may also be more generally true, because the sorted list $p\!-\!<\ x$ could be only one of many alternatives.

`all`

This function takes a predicate p to a predicate that holds if p is is true of every item of its argument. It is similar to the `g` pseudo-pointer (page 70).

all_same

This function takes any function f as an argument, not necessarily a predicate, and constructs a predicate that is true if f yields the same value when applied to every item of the input list. Note that this condition is stronger than logical equivalence, which implies only that two values are both empty or both non-empty, so care must be taken if f is a predicate whose true results may vary. This function is similar to the `K1` pseudo-pointer (page 84).

any

This function takes a predicate p as an argument, and returns a predicate that holds whenever p is true of at least one member of its input list. It is similar to the `k` pseudo-pointer (page 70).

8.8 Generalized set operations

The combinators documented in this section generalize the concepts of intersection, difference, and membership for lists and sets by parameterizing them with an arbitrary binary relational predicate.

gdif

This function takes a relational predicate p and returns a function that maps a pair of sets $(\{x_0 \dots x_n\}, \{y_0 \dots y_m\})$ to a copy of the left one with all x_i deleted for which there exists a y_j satisfying $p(x_i, y_j)$. The standard set difference operation is obtained with p as equality.

gint

This function takes a relational predicate p and returns a function that maps a pair of sets $(\{x_0 \dots x_n\}, \{y_0 \dots y_m\})$ to a copy of the left one with all x_i deleted for which there exists no y_j satisfying $p(x_i, y_j)$. The standard set intersection operation is obtained with p as equality.

gldif

This function follows the same calling convention as `gdif`, but constructs a function that operates on pairs of lists rather than pairs of sets by taking the order and multiplicity of the items into account. For each deleted x_i, a distinct y_j satisfies $p(x_i, y_j)$. A unique result is obtained by choosing the assignment of matching y's to deletable x's in the order they are detected by scanning forward through the y's for each x.

A short example using this function is the following.

```
$ fun --m="gldif~&E/'aaabbbcccaaa' 'aaccccd'" --c %s
'abbbaaa'
```

glint

This function performs an analogous operation to the generalized list difference combinator `gldif`, but pertains to intersection rather than difference.

The generalized set operations above are related to the `K10` through `K13` pseudo-pointers, whereas the remaining one is similar to the `w` pseudo-pointer or `-=` operator.

lsm

Given a set s, this function, mnemonic for “large set membership”, constructs a predicate that is true for all members of s and false otherwise.

Although it would be trivial to implement `lsm` as `\/-=`, the implementation in the standard library attempts to construct the optimal decision procedure for a large set, which may be more efficient than the default set membership algorithm of sequential search. The crossover point between the speed of the two algorithms for membership testing occurs around a cardinality of 8, not including the time required by `lsm` to construct the predicate. Best performance is achieved when the set members have most dissimilar representations.

I'm your number one fan.

Kathy Bates in *Misery*

9
Natural numbers

The natural numbers $0, 1, 2 \ldots$, are a primitive type in the language, with the type expression mnemonic `%n`, as explained in Chapter 3. Any application involving natural numbers may elect to manipulate them directly on the bit level. Alternatively, the `nat` module presents an interface to them as an abstract type.

Similarly to the `std` library documented in the previous chapter, the `nat` library is automatically loaded by the compiler's wrapper script, and need not be specified on the command line. This chapter documents its functions.

9.1 Predicates

A couple of functions take natural numbers as input and return a truth value.

nleq

This function computes the partial order relational predicate. Given a pair of numbers (n, m), it returns a non-empty value if and only if $n \leq m$.

An example using this function is the following.

```
$ fun --m="nleq* <(1,2),(4,3),(5,5)>" --c %bL
<true,false,true>
```

odd

This function returns a true value if and only if its argument is an odd number (i.e., $1, 3, 5 \ldots$).

9.2 Unary

The following functions take a natural number as an argument and return a natural number as a result.

- Standard mathematical notation is used in the descriptions (e.g., $n+1$) as opposed to language syntax in the examples (e.g., `double+ half`).
- Natural numbers in Ursala have unlimited precision, so overflow is not an issue for any of these functions unless the whole host machine runs out of memory.

half

This function performs truncating division by two. That is, given a number n, it returns $n/2$ if n is even, and returns $(n-1)/2$ if n is odd.

Half of the first six natural numbers are computed as follows.

```
$ fun --m="half* <0,1,2,3,4,5>" --c %nL
<0,0,1,1,2,2>
```

factorial

This function returns the factorial of an argument n, which is defined as $\prod_{i=1}^{n} i$, and has applications in combinatorial problems as the number of possible orderings of a sequence of n distinct items.

The factorial of a number n is conventionally denoted $n!$, but the exclamation point has an unrelated meaning in the language as the constant combinator.

double

Given a number n, this function returns the number $2n$.

The `double` function is a partial inverse to `half`, because `half+ double` is equivalent to the identity function. The function `double+ half` is equivalent to rounding down to the nearest even number.

predecessor

Given a number n, this function returns $n-1$ if $n > 0$, and raises an exception if $n = 0$. The diagnostic message in the latter case is "`natural out of range`".

successor

Given a number n, this function returns $n+1$.

`tenfold`

Given a number n, this function returns $10n$ by a fast bit manipulation algorithm.

9.3 Binary

All of the functions documented in this section take a pair of natural numbers as input. The `division` function returns a pair of natural numbers as a result, and the rest return a single natural number.

`sum`

This function takes a pair (n, m) to its sum $n + m$.

`difference`

This function takes a pair (n, m) to $n - m$ if $n \geq m$, but raises an exception if $n < m$. The diagnostic message in the latter case is "`natural out of range`".

`quotient`

This function takes a pair (n, m) and returns the quotient rounded down to the nearest natural number, $\lfloor n/m \rfloor$ unless $m = 0$. In that case, it raises an exception with the diagnostic message "`natural out of range`".

This example shows an exact and a truncated quotient.

```
$ fun --m="quotient* <(21,3),(100,8)>" --c %nL
<7,12>
```

`remainder`

This function takes a pair (n, m) and returns their residual, customarily denoted $n \bmod m$. This number is the remainder left over when n is divided by m, i.e., $((n/m) - \lfloor n/m \rfloor) \times m$.

The standard relationships between truncated quotients and residuals holds exactly.

```
^\~&r sum^/remainder product^/~&r quotient
```

This expression is equivalent to the identity function for a pair of natural numbers (n, m) provided $m \neq 0$.

`product`

This function multiplies a pair of numbers (n, m) to obtain their product nm.

`division`

The quotient and remainder can be obtained at the same time by this function more efficiently than computing them separately. Given a pair of number (n, m) with $m \neq 0$, this function returns a pair (q, r) where q is the quotient and r is the remainder.

The following identities hold.

$$\begin{aligned} \texttt{division} &\equiv \texttt{\^{}/quotient remainder} \\ \texttt{quotient} &\equiv \texttt{\~{}\&l+ division} \\ \texttt{remainder} &\equiv \texttt{\~{}\&r+ division} \end{aligned}$$

`choose`

Given a pair of natural numbers (n, m), this function returns the number of ways m elements can be selected from a set of n. This quantity is customarily denoted and defined as shown.

$$\binom{n}{m} = \frac{n!}{m!(n-m)!}$$

`gcd`

This function takes a pair (n, m) and returns their greatest common divisor, as obtained by Euclid's algorithm. The greatest common divisor is defined as the largest number k for which $(n \mod k) = (m \mod k) = 0$.

`root`

This function takes a pair (y, n) to the truncated n-th root of y, or $\lfloor \sqrt[n]{y} \rfloor$, using an iterative interval halving algorithm. If $n = 0$, y must be 1, or else an exception is raised with the diagnostic message "`zeroth root of non-unity`".

`power`

Given a pair of numbers (n, m) this function returns n^m, i.e., the product of n with itself m times.

This example shows the size of a conventional DES key space.

```
$ fun --m="power/2 56" --c
```

```
72057594037927936
```

However, powers of two are more efficiently obtained by bit shifting.

9.4 Lists

A couple of other functions in the `nat` library are useful for converting between numbers and lists.

iota

This function takes a natural number n and returns the list of n numbers from 0 to $n-1$ in ascending order.

This example shows how to generate the list of numbers from zero to fifteen.

```
$ fun --m=iota16 --c
<0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15>
```

nrange

This function takes a pair of natural numbers (a, b) and returns the list of natural numbers from a to b inclusive. If $b > a$, the list is given in descending order.

```
$ fun --m="nrange(3,19)" --c %nL
<3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19>
$ fun --m="nrange(19,3)" --c %nL
<19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3>
```

length

Given any list or set, this function returns its length or cardinality, respectively.

The following equivalence holds for any natural number n.

$$n = \texttt{length iota } n$$

Because natural numbers are represented as lists of booleans, they also have a length. Although there is no logarithm function defined in the `nat` library, a tight upper bound on the logarithm of a natural number to the base 2 can be found by taking its length.

```
$ fun --m="length factorial 52" --c %n
226
```

This result is confirmed by a more precise calculation using floating point arithmetic.

```
$ fun --m="..log2 ..nat2mp factorial 52" --c %E
2.255810E+02
```

He is you, your opposite, your negative, the result of the equation trying to balance itself out.

The Oracle in *The Matrix Revolutions*

10 Integers

Numbers like $\cdots -2, -1, 0, 1, 2\ldots$ of type `%z` are supported by operations in the `int` library documented in this chapter. Non-negative integers are binary compatible with natural numbers (type `%n`), and any of the functions described in this chapter will also work on natural numbers, albeit with the unnecessary overhead of checking their signs, which is not a constant time operation due to the representation used.

10.1 Notes on usage

Many functions in this chapter have the same names as similar functions in the `nat` library documented in the previous chapter. Using both in the same source text is possible by methods described in Section 7.2 to control the scope and visibility of imported symbols. For example, a file containing the directives

```
#import nat
#import int
```

in that order preceding any declarations will use integer functions by default, reverting to natural functions such as `iota` only when there is no integer equivalent, or when it is specifically requested using the dash operator, as in `nat-successor`. The opposite order will cause natural functions to be used by default unless otherwise indicated. Alternatively, integer operations can be used exclusively by using only the `#import int` directive and omitting `#import nat` from the source text.

10.2 Predicates

This section is for functions that return a boolean value when operating on integers.

`zleq`

This function computes the partial order relational predicate. Given a pair of numbers (n, m), it returns a non-empty (i.e., true) value if and only if $n \leq m$.

10.3 Unary Operations

The functions documented in this section take a single integer argument to an integer result.

`abs`

This function returns the absolute value of its argument. If the argument is non-negative, the result is the same as the argument. Otherwise, the result is its additive inverse. Hence, the result is always non-negative.

`sgn`

This function returns -1, 0, or 1, depending on whether its argument is negative, zero, or positive, respectively.

`negation`

This function returns the additive inverse of its argument. Negative numbers map to positive results, positives map to negatives, and zero to itself.

`successor`

Given any integer n, this function returns $n + 1$.

`predecessor`

Given any integer n, this function returns $n - 1$.

Unlike the `nat-predecessor` function, this one is defined for all integers.

10.4 Binary Operations

The functions documented in this section take a pair of integers as an argument and return an integer as a result.

sum

Given a pair (n, m) this function returns their sum, $n + m$.

difference

Given a pair (n, m) this function returns their difference, $n - m$.

Unlike the `nat-difference` function, this one is defined for all integers.

product

Given a pair (n, m) this function returns their product, nm.

quotient

Given a pair (n, m) with $m \neq 0$, this function returns $\lfloor n/m \rfloor$ if $n/m \geq 0$, and $\lceil n/m \rceil$ otherwise (i.e., the truncation toward zero of n/m).

The quotient rounding convention has been chosen to satisfy this identity.

$$\mathtt{abs}(\mathtt{quotient}(n, m)) \equiv \mathtt{quotient}(\mathtt{abs}(n), \mathtt{abs}(m))$$

remainder

Given a pair of integers (n, m) with $m \neq 0$ this function returns an integer r satisfying $\mathtt{sum}(\mathtt{product}(\mathtt{quotient}(n, m), m), r) = n$.

10.5 Multivalued

Function documented in this section return something other than a boolean or integer value.

division

This function maps a pair (n, m) of integers with $m \neq 0$ to the pair of integers $(\mathtt{quotient}(n, m), \mathtt{remainder}(n, m))$.

The same relationship among the `division`, `quotient`, and `remainder` functions holds for integers as for natural numbers. If both the quotient and remainder are required, it is more efficient to compute them using the division function than individually.

`zrange`

Given a pair of integers (n, m), this function returns the list of $|n - m + 1|$ integers beginning with n, ending with m and differing by 1 between consecutive items. If $n > m$, the numbers are listed in descending order.

For him, it's as if there were thousands of bars and behind the thousands of bars no world.

Robin Williams in *Awakenings*

11
Binary converted decimal

The type `%v` represents integers sequences of decimal digits, along with a boolean sign, as described on page 119, which may be more efficient than the usual binary representation in applications needing to manipulate and display numbers with thousands of digits or more. Literal numerical constants in this representation are written as sequences of decimal digits with a trailing underscore, and an optional leading negative sign.

A small set of functions for operating on numbers in this representation with a similar API to the `int` library described in the previous chapter is provided by the `bcd` library documented in this chapter. Because many of the functions are similarly named, the discussion of name clash resolution in Section 10.1 is relevant here as well.

11.1 Predicates

A partial order relational predicate on BCD integers is provided as follows.

bleq

This function computes the partial order relational predicate. Given a pair of numbers (n, m) in BCD format, it returns a non-empty (i.e., true) value if and only if $n \leq m$.

Here is an example usage.

```
$ fun bcd --m="^A(~&,bleq)*p 50%vi~*iiX 15" --c %vWbAL
<
   (-693480964_,6180548644_): true,
   (6597127700_,-532915486_): false,
   (-855627074_,-166599056_): true,
   (913347791_,8147630828_): true>
```

odd

This function returns a true value if its argument is not a multiple of 2, and a false value otherwise.

11.2 Unary Operations

The functions documented in this section take a single BCD argument to an BCD result.

abs

This function returns the absolute value of its argument. If the argument is non-negative, the result is the same as the argument. Otherwise, the result is its additive inverse. Hence, the result is always non-negative.

sgn

This function returns -1_, 0_, or 1_, depending on whether its argument is negative, zero, or positive, respectively.

Here are some examples.

```
$ fun bcd --m="^A(~&,sgn)* :/0_ 50%vi* 7" --c %vvAL
<
   0_: 0_,
   -3741541087_: -1_,
   306278996_: 1_,
   -12120849714_: -1_>
```

negation

This function returns the additive inverse of its argument. Negative numbers map to positive results, positives map to negatives, and zero to itself.

successor

Given any BCD integer n, this function returns $n + 1$.

predecessor

Given any BCD integer n, this function returns $n - 1$.

`tenfold`

This function returns its argument multiplied by ten, obtained using the obvious optimization in place of multiplication.

`factorial`

This function returns the factorial function a non-negative argument n, defined as $\prod_{i=1}^{n} i$.

11.3 Binary Operations

The functions documented in this section take a pair of BCD integers as an argument and return a BCD integer as a result.

`sum`

Given a pair (n, m) this function returns their sum, $n + m$.

`difference`

Given a pair (n, m) this function returns their difference, $n - m$.

`product`

Given a pair (n, m) this function returns their product, nm.

`quotient`

Given a pair (n, m) with $m \neq 0$, this function returns $\lfloor n/m \rfloor$ if $n/m \geq 0$, and $\lceil n/m \rceil$ otherwise (i.e., the truncation toward zero of n/m).

The quotient rounding convention has been chosen to satisfy this identity.

$$\texttt{abs}(\texttt{quotient}(n, m)) \equiv \texttt{quotient}(\texttt{abs}(n), \texttt{abs}(m))$$

`remainder`

Given a pair of integers (n, m) with $m \neq 0$ this function returns an integer r satisfying $\texttt{sum}(\texttt{product}(\texttt{quotient}(n, m), m), r) = n$.

`power`

Given a pair of BCD integers (n, m) with $m \geq 0$, this function returns the exponentiation n^m. Negative values of n are allowed, and will imply a negative result if m is odd. Zero raised to the power of zero is defined as $1\_$.

11.4 Multivalued

Function documented in this section return something other than a boolean or BCD value.

`division`

This function maps a pair (n, m) of integers with $m \neq 0$ to the pair of integers $(\texttt{quotient}(n, m), \texttt{remainder}(n, m))$.

The same relationship among the `division`, `quotient`, and `remainder` functions holds for BCD integers as for binary integers and natural numbers. If both the quotient and remainder are required, it is more efficient to compute them using the division function than individually.

`brange`

Given a pair of BCD integers (n, m), this function returns the list of $|n - m + 1|$ BCD integers beginning with n, ending with m and differing by 1 between consecutive items. If $n > m$, the numbers are listed in descending order.

11.5 Conversions

A couple of functions are defined provided for converting between BCD integers and other types.

`toint`

Given a BCD integer n, this function returns the corresponding integer in the binary representation (i.e., type `%z`, or if non-negative, type `%n`).

`fromint`

Given a natural number or integer in the binary representation (i.e., type `%n` or `%v`), this function returns the corresponding number converted to the BCD integer representation.

Don't knock rationalizations.

Jeff Goldblum in *The Big Chill*

12 Rational numbers

The primitive type `%q` represents rational numbers in unlimited precision. They can be used to perform exact numerical calculations with the functions defined in the `rat` library and documented in this chapter. Simultaneously their greatest strength and their greatest weakness, their exactitude renders them prohibitively inefficient for routine work, but they may be useful in special circumstances such as proof checking or conjecture.

12.1 Unary

The functions documented in this section take a single rational number as an argument to a rational result.

inverse

This function takes a number x to $1/x$.

This example shows inverses of two numbers.

```
$ fun rat --m="inverse* <5/2,-3/8>" --c %qL
<2/5,-8/3>
```

negation

This function takes any number x to $-x$.

In this example, a number is negated.

```
$ fun rat --m="negation 1/2" --c %q
-1/2
```

abs

This function returns the absolute value of its argument. That is, `abs` x is equal to x if x is positive but $-x$ if x is negative.

The following example shows absolute values of positive and a negative number.

```
$ fun rat --m="abs* <1/3,-2/5>" --c %qL
<1/3,2/5>
```

simplified

This function reduces a rational number to lowest terms. It is unnecessary for numbers computed by other functions in the library, but may be helpful for user defined functions.

The rational number representation consists of a pair of integers

$$(\langle \textit{numerator} \rangle, \langle \textit{denominator} \rangle)$$

which a user program may elect to construct directly. Following this operation with the `simplified` function will ensure that the representation meets the required invariant of being in lowest terms with a non-negative denominator.

```
$ fun rat --m="(2,4)" --c %q
fun: writing `core'
warning: can't display as indicated type; core dumped
$ fun rat --m="%qP (2,4)" --s
2/4
$ fun rat --m="simplified (2,4)" --c %q
1/2
```

12.2 Binary

The functions documented in this section take a pair of rational numbers and return a rational number, except for `rleq`, which returns a boolean value.

rleq

This function computes the partial order relation on rational numbers. Given a pair of numbers (x, y), it returns a true value if and only of $x \leq y$.

sum

This function takes a pair of numbers (x, y) to their sum $x + y$.

difference

This function takes a pair of numbers (x, y) to their difference $x - y$.

quotient

This function takes a pair of numbers (x, y) to the their quotient x/y.

product

This function takes a pair of numbers (x, y) to their product xy.

power

This function takes a pair of numbers (x, y) to their exponentiation x^y if this number is rational, but returns an empty value `()` otherwise.

Here are two examples of the `power` function, the second case having an irrational result.

```
$ fun rat --m="rat-power(27/8,4/3)" --c %qZ
81/16
$ fun rat --m="rat-power(27/8,2/5)" --c %qZ
()
```

12.3 Formatting

The functions documented in this section convert rational numbers to a character string representation compatible with the syntax of floating point numbers. In some cases, the string representation may require rounding. Each function takes a natural number as an argument specifying the number of decimal places, and returns a function that takes rational numbers to lists of strings.

fixed

This function takes a natural number n to a function that converts a rational number to a list of strings in fixed decimal format with n places after the decimal point.

scientific

This function takes a natural number n to a function that converts a rational number to a list of strings in exponential notation with n places after the decimal point.

engineering

This function takes a natural number n to a function that converts a rational number to a list of strings in exponential notation with $n + 1$ decimal places and the exponent chosen to be a multiple of 3.

Here are examples of the same number in all three formats.

```
$ fun rat --m="engineering4 35737875/131" --s
272.80e+03
$ fun rat --m="scientific4 35737875/131" --s
2.7280e+05
$ fun rat --m="fixed4 35737875/131" --s
272808.2061
```

Logsine, clogsine, thingamabob, some bubblegum will do the job.

The Nowhere Man in *Yellow Submarine*

13

Floating point numbers

Ursala places substantial resources at the developer's disposal in the way of floating point number operations. A small library, `flo`, containing some of the more frequently used functions and constants is documented in this chapter. Other libraries pertaining to more specialized areas are documented in subsequent chapters, and these are further augmented by the virtual machine's interface to third party numerical libraries as documented in the `avram` reference manual.

All functions described in this chapter involve floating point numbers in standard IEEE double precision format, corresponding to the primitive type `%e` in the language. Users interested in arbitrary precision numbers (type `%E`) are referred to the documentation of the `mpfr` library in the `avram` reference manual, whose functions are directly accessible by the library combinators (Section 6.7.2, page 216).

13.1 Constants

The declarations documented in this section pertain to numerical constants. These are usable as numbers in expressions, and require not much further explanation.

`eps`

A small number on the order of the machine precision, arbitrarily defined as 5×10^{-16}.

`inf`

A constant having the algebraic properties of infinity (∞), such as $x/\infty = 0$ for finite x, *etcetera*.

`nan`

A constant representing an indeterminate result, such as $\infty - \infty$, which will propagate automatically through any computation depending on it.

The representation of indeterminate results is not unique, so it is not valid to test a result for indeterminacy by comparing it to `nan`. The predicate `math..isnan` should be used instead for that purpose.

`ninf`

A constant having the algebraic properties of negative infinity, $-\infty$, analogous to the `inf` constant explained above.

`pi`

The mathematical constant $3.14159\ldots$ familiar from trigonometry

13.2 General

General unary and binary operations on floating point numbers are documented in this section. Most of them are simple wrappers for the corresponding virtual machine `math..` library functions, defined as a matter of convenience.

13.2.1 Unary

The following functions take a single floating point number as an argument and return a floating point number as a result.

`abs`

The absolute value function, customarily denoted $|x|$ for an argument x, returns x if x is positive or zero, and $-x$ otherwise.

`negative`

This function takes an argument x to its additive inverse, $-x$.

`sqr`

This function takes a number x and returns x^2.

`sqrt`

This function takes a number x and returns $\sqrt{x}$. The result is `nan` if $x < 0$.

`sgn`

This function takes any argument to a result of -1, 0, or 1, depending on whether the argument is negative, zero, or positive, respectively. The IEEE standard admits a notion of -0, which is considered negative by this function.

13.2.2 Binary

The usual binary operations on floating point numbers are provided by the functions documented in this section. Each of them takes a pair of numbers as input and returns a number as a result. Correct handling of indeterminate (`nan`) and infinite arguments is automatic. Overflowing results are mapped to infinity.

`plus`

Given a pair (x, y), this function returns the sum, $x + y$.

`minus`

Given a pair (x, y), this function returns the difference $x - y$.

`times`

Given a pair (x, y) this function returns the product, xy.

`div`

Given a pair (x, y), this function returns the quotient x/y. A result of `nan` is possible if y is 0.

`pow`

Given a pair (x, y), this function returns the exponentiation x^y if it is representable without overflow.

bus

Given a pair (x, y) this function returns the difference $y - x$, i.e., with the order reversed.

vid

Given a pair (x, y), this function returns the quotient y/x.

The last two functions are often more convenient than the conventional forms of subtraction and division. For example, to subtract the baseline from a list of floating point numbers, it is slightly quicker and less cluttered to write

```
bus^*D\~& fleq$-
```

than the alternative

```
sub^*DrlXS\~& fleq$-
```

13.3 Relational

The following functions involve tests or comparisons on floating point numbers.

fleq

This function computes the partial order relation on floating point numbers, returning a true value if and only if a given pair of numbers (x, y) satisfies $x \leq y$. The predicate does not hold if either number is indeterminate.

max

Given a pair of numbers (x, y), this function returns y if $y \geq x$, and returns x otherwise. A `nan` value isn't greater or equal to anything.

min

Given a pair of numbers (x, y), this function returns x if $x \leq y$, and returns y otherwise.

zeroid

This function returns a true value if its argument is exactly 0. Negative 0 is also considered zero, but small values differing from zero by representable roundoff error are not.

13.4 Trigonometric

Wrappers for circular functions provided by the virtual machine's `math..` library are defined for convenience as shown below. Each of these functions takes a floating point argument to a floating point result. The inverse functions may return a `nan` value for arguments outside their domains.

`sin`

This function returns the sine of a given number x.

`cos`

This function returns the cosine of a given number x.

Definitions of sine and cosine functions are given by the standard construction involving the unit circle.

`tan`

This function returns the tangent of a given number x, which can be defined as $\sin(x)/\cos(x)$.

`asin`

Given a number y, this function returns an x satisfying $y = \sin(x)$ if possible.

`acos`

Given a number y, this function returns an x satisfying $y = \cos(x)$ if possible.

`atan`

Given a number y, this function returns an x satisfying $y = \tan(x)$ if possible.

13.5 Exponential

A short selection of functions pertaining to exponents and logarithms is provided as described below. Each of these functions takes a single floating point argument to a floating point result.

exp

Given a number x, this function returns the exponentiation e^x, where e is the standard mathematical constant $2.71828\ldots$.

ln

For a positive number x, this function returns the natural logarithm $\ln x$, which can be defined as the number y satisfying $x = e^y$.

tanh

This is the so called hyperbolic tangent function, which is defined as

$$\tanh(x) = \frac{e^x - e^{-x}}{e^x + e^{-x}}$$

atanh

Given a number y between -1 and 1, this function returns a number x satisfying $y = \tanh(x)$.

13.6 Calculus

Several higher order functions supporting elementary operations from integral and differential calculus are provided as documented in this section.

derivative

Given a real valued function f of a single real variable, this function returns another function f', which is pointwise equal to the instantaneous rate of change of f.

This function works best for smooth continuous functions f. The function is differentiated numerically by the GNU Scientific Library numerical differentiation routine with the central difference method. Users requiring the forward or backward difference (for example to differentiate a function at 0 that is defined only for non-negative input) can use the GSL functions directly as documented by the `avram` reference manual.

A short example of this function shows how $f(x) = x^2$ can be differentiated, and the resulting function sampled over a range of input values, using the `ari` function documented subsequently in this chapter to generate an arithmetic progression of eleven values for x ranging from zero to one.

```
$ fun flo --m="^(~&,derivative sqr)* ari11/0. 1." --c %eWL
```

```
<
   (0.000000e+00,0.000000e+00),
   (1.000000e-01,2.000000e-01),
   (2.000000e-01,4.000000e-01),
   (3.000000e-01,6.000000e-01),
   (4.000000e-01,8.000000e-01),
   (5.000000e-01,1.000000e-00),
   (6.000000e-01,1.200000e+00),
   (7.000000e-01,1.400000e+00),
   (8.000000e-01,1.600000e+00),
   (9.000000e-01,1.800000e+00),
   (1.000000e+00,2.000000e+00)>
```

For each value of x, the derivative of $f(x)$ is $2x$, as expected.

nth_deriv

This function takes a natural number n to a function that returns the n-th derivative of a given function f.

The function `nth_deriv1` is equivalent to the `derivative` function. Ideally the function `nth_deriv2` would be equivalent to `derivative+ derivative`, and so on, but in practice there are problems with numerical stability when taking higher derivatives. The `nth_deriv` function attempts to obtain better results than the naive approach by using an ensemble of progressively larger tolerances for the higher derivatives when invoking the underlying GSL differentiation routine.

integral

Given a function f taking a real value to a real result, this function returns a function F taking a pair of real values to a real result, such that

$$F(a,b) = \int_{x=a}^{b} f(x)\ \mathrm{d}x$$

The following examples demonstrate the `integral` function.

```
$ fun flo --m="integral(sqr)/0. 3." --c %e
9.000000e+00
$ fun flo --m="integral(sin)/0. pi" --c %e
2.000000e+00
```

The `integral` function is based on the GNU Scientific Library integration routines, using the adaptive algorithm iterated over a range of tolerances if necessary. This function will give best results in most cases, but users requiring more specific control (e.g., to specify tolerances or discontinuities explicitly) are referred to the `avram` reference manual for information on how to access these features.

root_finder

This function takes a quadruple $((a, b), (f, t))$ where f is a real valued function of a real variable and the other parameters are real. It returns a floating point number x such that $a \leq x \leq b$ and $|x - x_0| \leq t$, where $f(x_0) = 0$. If no such x exists, the result is unspecified.

The function finds a root by a simple bisection algorithm. The algorithm guarantees convergence subject to machine precision if there is a unique root on the interval, but doesn't converge as fast as more sophisticated methods based on stronger assumptions. The following example retrieves a root of the sine function between 3 and 4. The exact solution is of course π.

```
$ fun flo --m="root_finder((3.,4.),(sin,1.e-8))" --c %e
3.141593e+00
```

13.7 Series

The functions documented in this section are useful for operating on vectors or time series represented as lists of floating point numbers.

13.7.1 Accumulation

These three functions perform cumulative operations, each taking a list of numbers as input to a list of numbers as output. Differences are inverses of cumulative sums.

cu_prod

Given a list $\langle x_0 \dots x_n \rangle$ this function returns the list $\langle y_0 \dots y_n \rangle$ for which

$$y_i = \prod_{j=0}^{i} x_j$$

.

Here is a simple example of a cumulative product.

```
$ fun flo --m="cu_prod <1.,2.,3.,4.,5.>" --c
<
   1.000000e+00,
   2.000000e+00,
   6.000000e+00,
   2.400000e+01,
   1.200000e+02>
```

cu_sum

Given a list $\langle x_0 \dots x_n \rangle$ this function returns the list $\langle y_0 \dots y_n \rangle$ for which

$$y_i = \sum_{j=0}^{i} x_j$$

.

Here is a simple example of a cumulative sum.

```
$ fun flo --m="cu_sum <1.,2.,3.,4.,5.,6.,7.,8.,9.>" --c
<
   1.000000e+00,
   3.000000e+00,
   6.000000e+00,
   1.000000e+01,
   1.500000e+01,
   2.100000e+01,
   2.800000e+01,
   3.600000e+01,
   4.500000e+01>
```

nth_diff

This function takes a natural number n to a function that computes the n-th difference of a list of numbers. For a given list of numbers $\langle x_1 \dots x_m \rangle$, the n-th difference is the list of numbers $\langle y_0^n \dots y_{n-m}^n \rangle$ satisfying this recurrence.

$$\begin{array}{rcl} y_i^0 & = & x_i \\ y_i^n & = & y_{i+1}^{n-1} - y_i^{n-1} \end{array}$$

The n-th difference requires the input list to have more than n items, because it get shortened by n. Here are three examples.

```
$ fun flo --m="nth_diff1 <2.,8.,7.,1.>" --c
<6.000000e+00,-1.000000e+00,-6.000000e+00>
$ fun flo --m="nth_diff2 <2.,8.,7.,1.>" --c
<-7.000000e+00,-5.000000e+00>
$ fun flo --m="nth_diff3 <2.,8.,7.,1.>" --c
<2.000000e+00>
```

13.7.2 Binary vector operations

These two functions compute the standard metrics on pairs of vectors.

`iprod`

Given a pair of lists of floating point numbers $(\langle x_0 \dots x_n \rangle, \langle y_0 \dots y_n \rangle)$ having the same length, this function returns the inner product, which is defined as

$$\sum_{i=0}^{n} x_i y_i$$

`eudist`

Given a pair of lists of floating point numbers $(\langle x_0 \dots x_n \rangle, \langle y_0 \dots y_n \rangle)$ having the same length, this function returns the Euclidean distance between them, which is defined as

$$\sqrt{\sum_{i=0}^{n} (x_i - y_i)^2}$$

For vectors representing Cartesian coordinates of points in a flat two or three dimensional space, the Euclidean distance corresponds to the ordinary concept of distance between them as measured by a ruler. In data mining or pattern recognition applications, Euclidean distance is sometime useful as a measure of dissimilarity between a pair of time series or feature vectors.

`oprod`

Given a pair of lists of floating point numbers $(\langle x_0 \dots x_n \rangle, \langle y_0 \dots y_n \rangle)$ having the same length, this function returns a list $\langle z_0 \dots z_n \rangle$ of that length in which this relation holds.

$$z_i = \begin{cases} x_n y_1 - x_1 y_n & \text{if} \quad i = 0 \\ (-1)^n (x_{n-1} y_0 - x_0 y_{n-1}) & \text{if} \quad i = n \\ (-1)^i (x_{i-1} y_{i+1} - x_{i+1} y_{i-1}) & \text{otherwise} \end{cases}$$

If $n < 2$, the result is undefined.

This function computes the same outer product familiar from college physics, but generalizes it to higher dimensions. For example, the magnetic force exerted on a moving charged particle is proportional to the outer product of its velocity with the ambient magnetic field. In graphics applications, the outer product is an easy way to construct a vector that is perpendicular to the plane containing two given vectors.

13.7.3 Progressions

These two functions allow arithmetic or geometric progressions to be constructed without explicit iteration required.

`ari`

Given a natural number n, this function returns a function that takes a pair of floating point numbers (a, b) to a list $\langle x_1 \dots x_n \rangle$ of length n, wherein

$$x_i = a + \frac{(i-1)(b-a)}{n-1}$$

That is, there are n numbers at regular intervals starting from a and ending with b.

This example shows a list of four numbers from 25 to 40.

```
$ fun flo --m="ari4/25. 40." --c
<
   2.500000e+01,
   3.000000e+01,
   3.500000e+01,
   4.000000e+01>
```

`geo`

Given a natural number n this function returns a function that takes a pair of positive floating point numbers (a, b) to a list of n floating point numbers $\langle x_1 \dots x_n \rangle$ in geometric progression from a to b. That is,

$$x_i = a \exp\left(\frac{i-1}{n-1} \ln \frac{b}{a}\right)$$

The following example shows a geometric progression from 10 to 1000.

```
$ fun flo --m="geo5/10. 1000." --c
<
   1.000000e+01,
   3.162278e+01,
   1.000000e+02,
   3.162278e+02,
   1.000000e+03>
```

13.7.4 Extrapolation

These two functions can be used to extapolate a convergent series and thereby estimate the limit more efficiently than by direct computation.

`levin_limit`

Given a list of floating point numbers $\langle x_0 \dots x_n \rangle$, this function returns an estimate of the limit of x_n as n approaches infinity, based on the Levin-u transform from the GNU Scientific library.

This example shows the limit of a geometric series of numbers approaching 1.

```
$ fun flo --m="levin_limit <0.5,.75,.875,.9375>" --c
1.000000e-00
```

levin_sum

Given a list of floating point numbers $\langle x_0 \dots x_n \rangle$, this function returns an estimate of the limit of the sum of the series $\sum_{i=0}^{n} x_i$ as n approaches infinity.

This example shows the limit of the sum of a series of whose terms approach zero.

```
$ fun flo --m="levin_sum <0.5,.25,.125,.0625>" --c
1.000000e+00
```

13.8 Statistical

A selection of functions pertaining to statistics is documented in this section. These include descriptive statistics on populations, random number generators, and probability distributions.

13.8.1 Descriptive

The following functions compute standard moments and related parameters for data stored in lists of floating point numbers.

mean

Given a list of n numbers $\langle x_1 \dots x_n \rangle$, this function returns the population mean, defined as

$$\bar{x} = \frac{1}{n} \sum_{i=1}^{n} x_i$$

If the available data $\langle x_1 \dots x_n \rangle$ are a sample of the population rather than the whole population, a more statistically efficient estimator of the true mean has $n-1$ in the denominator rather than n. Users working with sample data may wish to define a different version of this function accordingly.

variance

For a list of numbers $\langle x_1 \dots x_n \rangle$, this function returns the variance, which is defined as

$$\frac{1}{n} \sum_{i=1}^{n} (x_i - \bar{x})^2$$

where $\bar{x}$ is the mean as defined as above.

stdev

This function returns the standard deviation of a list of numbers, which is defined as the square root of the variance.

covariance

Given a pair of lists of numbers $(\langle x_1 \dots x_n\rangle, \langle y_1 \dots y_n\rangle)$ of the same length n, this function returns the covariance, which is defined as

$$\frac{1}{n}\sum_{i=1}^{n}(x_i - \bar{x})(y_i - \bar{y})$$

In this expression, $\bar{x}$ is the mean of $\langle x_1 \dots x_n\rangle$ and $\bar{y}$ is the mean of $\langle y_1 \dots y_n\rangle$ as defined above.

correlation

This function takes a pair of lists of numbers to their correlation, which is defined as the covariance divided by the product of the standard deviations.

13.8.2 Generative

A couple of functions are defined for pseudo-random number generation. Strictly speaking they are not really functions because they may map the same argument to different results on different occasions.

rand

This function returns a pseudo-random number uniformly distributed between zero and one.

The following example shows five uniformly distributed pseudo-random numbers.

```
$ fun flo --m="rand* iota5" --c
<
   2.066991e-02,
   9.812020e-01,
   1.900977e-01,
   5.668466e-01,
   6.280061e-01>
```

The results are derived from the virtual machine's implementation of the Mersenne Twister algorithm, as documented in the `avram` reference manual.

Z

This function returns a pseudo-random number normally distributed with a mean of zero and a standard deviation of one. This distribution has a probability density function given by

$$\rho(x) = \frac{1}{\sqrt{2\pi}} \exp\left(-\frac{x^2}{2}\right)$$

Here are a few normally distributed random numbers.

```
$ fun flo --m="Z* iota3" --c
<7.760865e-01,2.605296e-01,-5.365909e-01>
```

This function depends on the virtual machine's interface to the `R` math library, which must be installed on host system in order for it to work.

13.8.3 Distributions

The functions described in this section provide cumulative and inverse cumulative probability densities. Currently only the standard normal distribution is supported, as defined above.

N

Given a number x, this function returns

$$\frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(-\frac{x^2}{2}\right) \,\mathrm{d}x$$

which is the probability that a random draw from a standard normal population will be less than x.

Q

Given a number y, this function returns a number x satisfying

$$y = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^{x} \exp\left(-\frac{x^2}{2}\right) \,\mathrm{d}x$$

It is therefore the inverse of the cumulative normal probability function defined above.

13.9 Conversion

Three functions allow conversions between floating point numbers and other types.

float

Given a natural number n of type `%n`, this function returns the equivalent of n in a floating point representation.

A simple example demonstrates this function.

```
$ fun flo --m=float125 --c
1.250000e+02
```

floatz

Given an integer n of type `%z`, this function returns the equivalent of n in a floating point representation.

Although natural numbers and positive integers have the same representation, the `floatz` function is necessary for coping with negative integers correctly. A negative argument to the `float` function will have an unspecified result.

strtod

This function takes a character string as input and returns a floating point number representation obtained by the `strtod` function from the host system's C library. The same syntax for floating point numbers as in C is acceptable. If the syntax is not valid, a value of floating point 0 is returned.

Here is an example of the `strtod` function.

```
$ fun flo --m="strtod '6.023e23'" --c
6.023000e+23
```

printf

This function takes a pair (f, x) as an argument. The left side f is a character string containing a C style format conversion for exactly one double precision floating point number, such as `'%0.4e'`, and the parameter x is a floating point number. The result returned is a character string expressing the number in the specified format.

Here is an example of the `printf` function being used to print π in fixed decimal format with five decimal places.

```
$ fun flo --m="printf/'%0.5f' pi" --c %s
'3.14159'
```

The higher I go, the crookeder it becomes.

Al Pacino in *The Godfather, Part III*

14 Curve fitting

A selection of functions in support of curve fitting or interpolation is provided in the `fit` library. These include piecewise polynomial and sinusoidal interpolation methods, available in both IEEE standard floating point and arbitrary precision arithmetic by way of the virtual machine's interface to the `mpfr` library. There are also functions for differentiation and higher dimensional interpolation.

The functions in this chapter are suitable for finding exact fits for data sets associating a unique output with each possible input. Readers requiring least squares regression or generalizations thereof may find the `lapack` library helpful, particularly the functions `dgelsd` and `dggglm`, which are conveniently accessible by way of the virtual machine's `lapack` interface as documented in the `avram` reference manual.

14.1 Interpolating function generators

The functions in this section take a set of points as an argment and return a function fitting through the points as a result.

plin

Given a set of pairs of floating point numbers $\{(x_0, y_0) \dots (x_n, y_n)\}$, this function returns a function f such that $f(x_i) = y_i$ for any (x_i, y_i) in the data set, and $f(x)$ is the linearly interpolated y value for any intermediate x.

Piecewise linear interpolation is an expedient method based on approximating the given function with connected linear functions. An illustration is given in Figure 14.1. Note that there is no requirement for the points to be equally spaced. The following example shows how the `plin` function can be used.

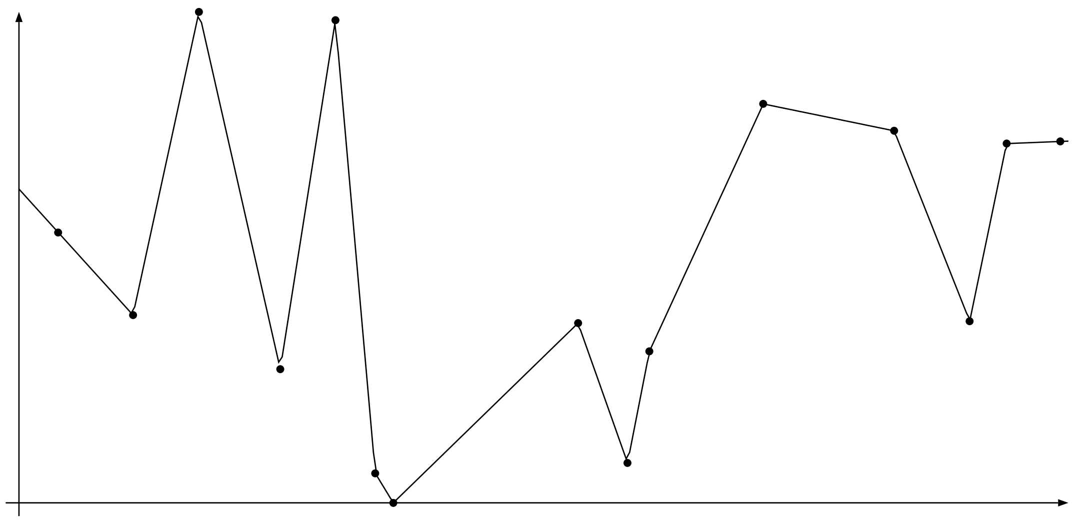


Figure 14.1: piecewise linear interpolation

```
$ fun flo fit --m="plin<(1.,2.),(3.,4.)>* ari5/1. 3." --c
<
   2.000000e+00,
   2.500000e+00,
   3.000000e+00,
   3.500000e+00,
   4.000000e+00>
```

sinusoid

Given a set of pairs of floating point numbers $\{(x_0, y_0) \dots (x_n, y_n)\}$, this function returns a function f such that $f(x_i) = y_i$ for any (x_i, y_i) in the data set, and $f(x)$ is the sinusoidally interpolated y value for any intermediate x.

mp_sinusoid

This function follows the same conventions as the `sinusoid` function, but uses arbitrary precision numbers in `mpfr` format as inputs and outputs.

For the latter function, The precision of numbers used in the calculations is determined by the precision of the numbers in the input data set.

As the names imply, these functions use a sinusoidal interpolation method. For equally spaced values of x_i, the function that they construct is evaluated by

$$f(x) = \sum_{i=0}^{n} y_i \frac{\sin(\omega(x - x_i))}{x - x_i}$$

for values of x other than x_i, with a suitable choice of ω.

- A function of this form has the property of being continuous and non-vanishing in all derivatives, and is also the minimum bandwidth solution.
- If the numbers x_i are not equally spaced, the spacing is adjusted by a cubic spline transformation to make this form applicable.
- Large variations in spacing may induce spurious high frequency oscillations or discontinuities in higher derivatives.

one_piece_polynomial

Given a set of pairs of floating point numbers $\{(x_0, y_0) \dots (x_n, y_n)\}$, this function returns a function f of the form

$$f(x) = \sum_{i=0}^{n} c_i x^i$$

with c_i chosen to ensure $f(x_i) = y_i$ for all (x_i, y_i) in the set.

mp_one_piece_polynomial

This function is the same as the one above except that it uses arbitrary precision numbers in `mpfr` format. The precision of numbers used in the calculations is determined by the input set.

With only two input points, the `one_piece_polynomial` degenerates to linear interpolation, as this example suggests.

```
$ fun fit -m="one_piece_polynomial{(1.,1.),(2.,2.)} 1.5" -c
1.500000e+00
```

However, for linear interpolation, the `plin` function documented previously is more efficient.

The polynomial interpolation function is obviously differentiable and arguably an aesthetically appealing curve shape, but it is prone to inferring extrema that are not warranted by the data, making it too naive a choice for most curve fitting applications.

14.2 Higher order interpolating function generators

The functions documented in this section allow for the construction of families of interpolating functions parameterized by various means. There is a piecewise polynomial interpolation method with selectable order similar to the conventional cubic spline method, a higher dimensional interpolation function, and a function for differentiation of polynomials obtained by interpolation.

chord_fit

This function takes a natural number n as an argument, and returns a function that takes a set of pairs of floating point numbers $\{(x_0, y_0) \dots (x_m, y_m)\}$ to a function f satisfying $f(x_i) = y_i$ for all points in the set. For other values of x, the function f returns a number y obtained by piecewise polynomial interpolation using polynomials of order $n + 3$ or less.

mp_chord_fit

This function is similar to the one above but uses arbitrary precision numbers in `mpfr` format. The precision of the numbers used in the calculations is determined by the precision of the numbers in the input data set.

The `chord_fit` functions generate functions f having the property that

$$f'(x_i) = \frac{f(x_{i+1}) - f(x_{i-1})}{x_{i+1} - x_{i-1}}$$

for the interior data points x_i, where f' is the first derivative of f. That is to say, the tangent to the curve at any given x_i from the data set is parallel to the chord passing through the neighboring points. Any additional degrees of freedom afforded by the order n are used to meet the analogous conditions for higher derivatives.

- Numerical instability imposes a practical limit of $n = 3$ for the fixed precision version.
- Higher orders are feasible for the arbitrary precision version provided that the numbers in the input list are of suitably high precision.
- There is unlikely to be any visually discernible difference in a plot of the curve for orders higher than 3.

A qualitative comparison of the three interpolation methods discussed hitherto is afforded by Figure 14.2. The figure includes one curve made by each method for the same randomly generated data set. The spline interpolation is made by the `chord_fit` function with a value of n equal to 0. It can be seen that the piecewise interpolation fits the data most faithfully, and is generally to be preferred for most data visualization or numerical work. The sinusoidal fit has a more wave-like appearance with symmetric peaks and troughs, of possible interest in signal processing applications. The one piece polynomial fit exhibits extreme fluctuations.

poly_dif

This function takes a natural number n as an argument, and returns a function that takes a function f as an argument to a function f'. The function f is required to be an interpolating function generated by either of the `one_piece_polynomial` or `chord_fit` functions. The function f' will be the n-th derivative of f.

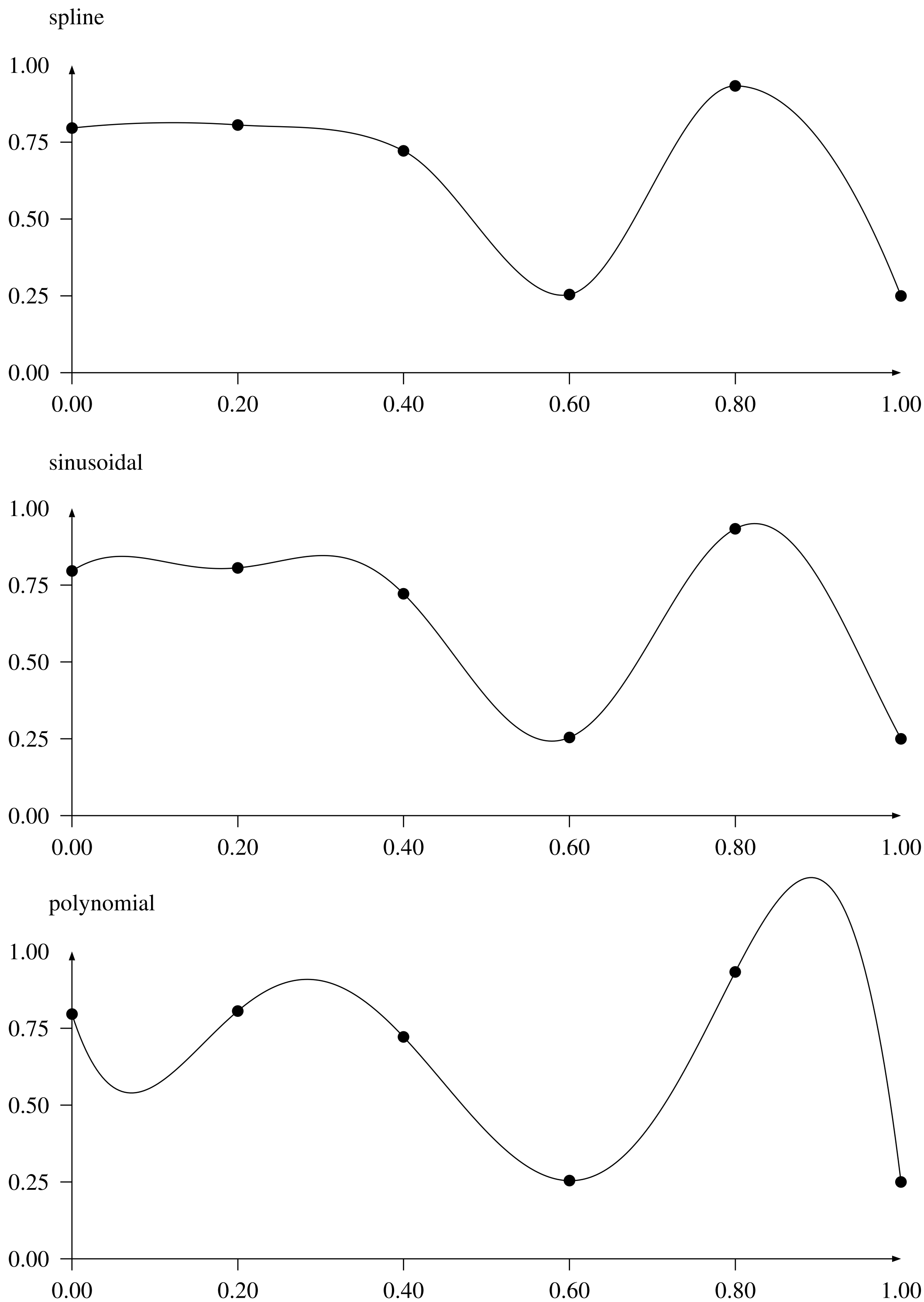


Figure 14.2: three kinds of interpolation

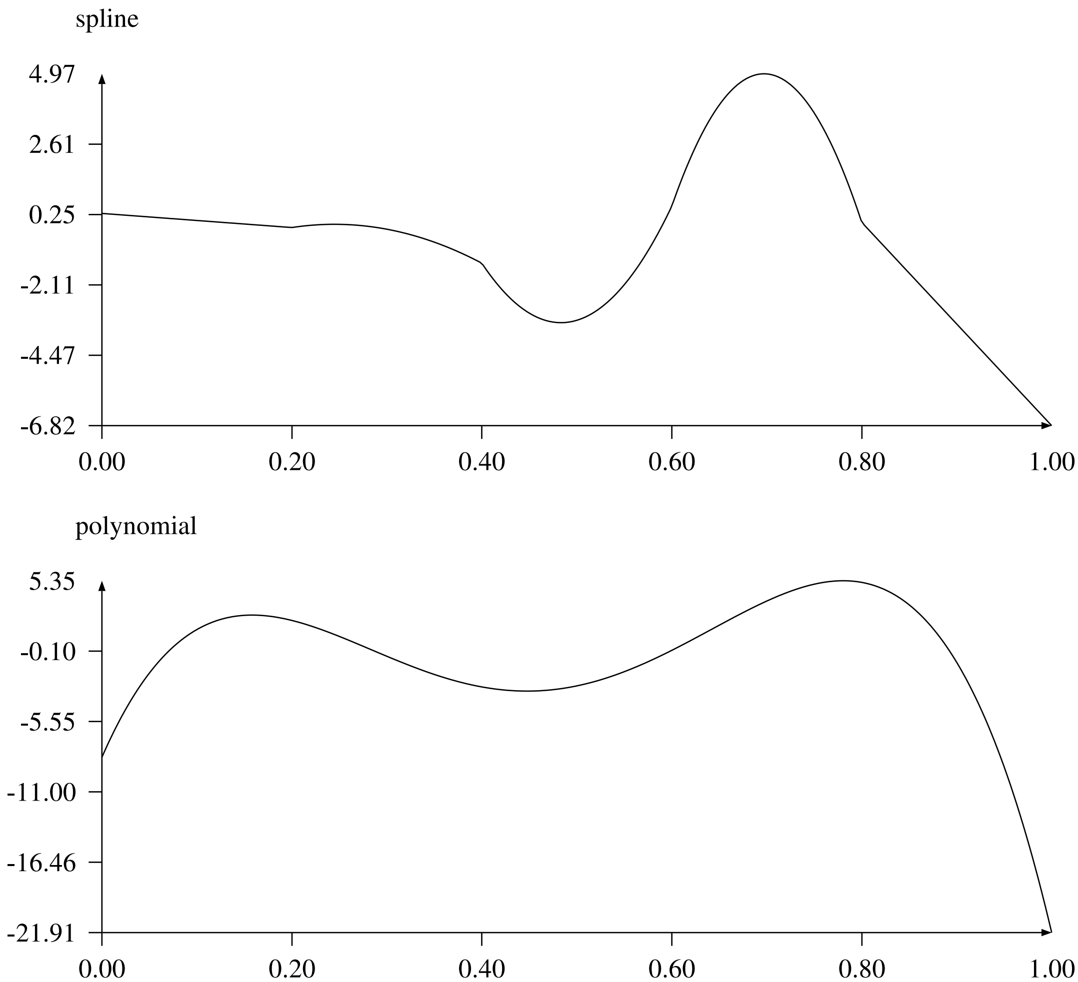


Figure 14.3: first derivatives of Figure 14.2 by the `poly_dif` function

The `poly_dif` function is specific to polynomial interpolating functions because it decompiles them based on the assumption that they have a certain form. The `derivative` function from the `flo` library can be used for differentiation in more general cases. However, differentiation by the `poly_dif` function is more accurate and efficient where possible.

Figure 14.3 shows plots of the first derivatives of the polynomial functions in Figure 14.2 as obtained by the `poly_dif` function. Figure 14.4 shows the same functions differentiated by the `derivative` function for comparison, as well as the first derivative of the sinusoidal interpolation.

- It can be noted from these figures that the piecewise interpolation is continuous but not smooth in the first derivative, and hence discontinuous in higher derivatives.

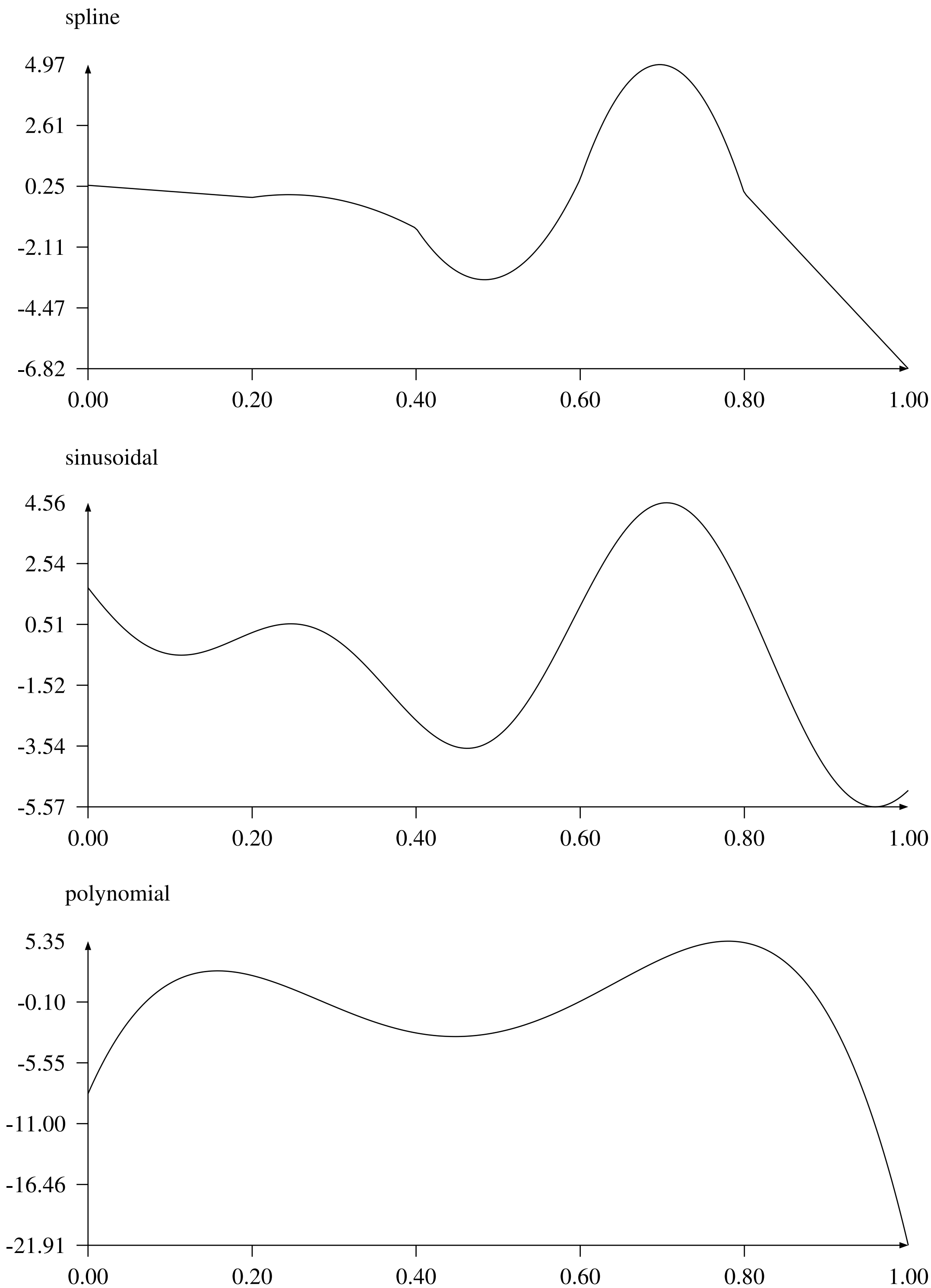


Figure 14.4: first derivatives of Figure 14.2 by the `flo-derivative` function

- The first and last intervals have linear first derivatives because only second degree polynomials are used there.

The interpolation methods described hitherto can be generalized to functions of any number of variables in a standard form by the higher order function described next. The function itself is meant to be parameterized by one of the generators (that is, `plin`, `sinusoid`, `mp_sinusoid`, `chord_fit` n, or `one_piece_polynomial`). It yields a generator taking points in a higher dimensional space specified by a lists of two or more input values per point.

multivariate

This function takes an interpolating function generator g for functions of one variable and returns an interpolating function generator G for functions of many variables.

- The input function g should take a set of pairs $\{(x_1, f(x_1)) \dots (x_n, f(x_n))\}$ as input, and return an interpolating function $\hat{f}$.
  - For x_i in the given data set, $\hat{f}(x_i) = f(x_i)$.
  - For other inputs z, a corresponding output is interpolated by $\hat{f}$.
- The output function G will take a set of lists as input,

$$\{\langle x_{11} \dots x_{1n}, F\langle x_{11} \dots x_{1n}\rangle\rangle \dots \langle x_{m1} \dots x_{mn}, F\langle x_{m1} \dots x_{mn}\rangle\rangle\}$$

  where $m = \prod_j |\bigcup_i \{x_{ij}\}|$, and return an interpolating function $\hat{F}$.
  - For lists of values $\langle x_{i1} \dots x_{in}\rangle$ in the given data set,

$$\hat{F}\langle x_{i1} \dots x_{in}\rangle = F\langle x_{i1} \dots x_{in}\rangle$$

  - For other inputs $\langle z_1 \dots z_n\rangle$, an output value is interpolated by $\hat{F}$.

Intuitively, the technical condition on m means that the interpolation function generator G depends on the assumption of the x_{ij} values forming a fully populated orthogonal array. For each j, there are

$$d_j = |\bigcup_i \{x_{ij}\}|$$

distinct values for x_{ij}. The number d_j can be visualized as the number of hyperplanes perpendicular to the j-th axis, or as the j-th dimension of the array. The product of d_j over j is the number of points required to occupy every position, hence the total number of points in the data set. A diagnostic message of "`invalid transpose`" may be reported if the data set does not meet this condition, or erroneous results may be obtained.

The interpolation algorithm can be explained as follows. If $n = 1$, the problem reduces to the one dimensional case. For interpolation in higher dimensions, it is solved recursively.

x	y	z
0.00	0.00	0.76476544
	1.00	0.91931626
	2.00	-2.60410277
	3.00	7.35946680
1.00	0.00	-5.05349099
	1.00	-4.06599595
	2.00	-1.02829526
	3.00	-8.83046108
2.00	0.00	0.91525110
	1.00	-4.08125924
	2.00	5.54509092
	3.00	5.68363915
3.00	0.00	2.60476835
	1.00	1.86059152
	2.00	-1.41751767
	3.00	-2.46337713

Table 14.1: randomly generated discrete bivariate function with inputs (x, y) and output z

- For each $X_k \in \bigcup_i \{x_{i1}\}$ with k ranging from 1 to d_1, a lower dimensional interpolating function f_k is constructed from the set of points shown below.

$$f_k = G\{\langle x_{12} \dots x_{1n}, F\langle X_k, x_{12} \dots x_{1n}\rangle\rangle \dots \langle x_{m2} \dots x_{mn}, F\langle X_k, x_{m2} \dots x_{mn}\rangle\rangle\}$$

- To interpolate a value of $\hat{F}$ for an arbitrary given input $\langle z_1 \dots z_n\rangle$, a one dimensional interpolating function h is constructed from this set of points

$$h = g\{(X_1, f_1\langle z_2 \dots z_n\rangle) \dots (X_{d_1}, f_{d_1}\langle z_2 \dots z_n\rangle)\}$$

and $\hat{F}\langle z_1 \dots z_n\rangle$ is taken to be $h(z_1)$.

Three small examples of two dimensional interpolation are shown in Figures 14.5 through 14.7. These surfaces are interpolated from the randomly generated data shown in Table 14.1. Figure 14.5 is generated by the function `multivariate chord_fit0`. Figure 14.6 is generated by `multivariate sinusoid`, and Figure 14.7 is generated by `multivariate one_piece_polynomial`. Qualitative differences in the shapes of the surfaces are commended to the reader's attention. Note that the vertical scales differ.

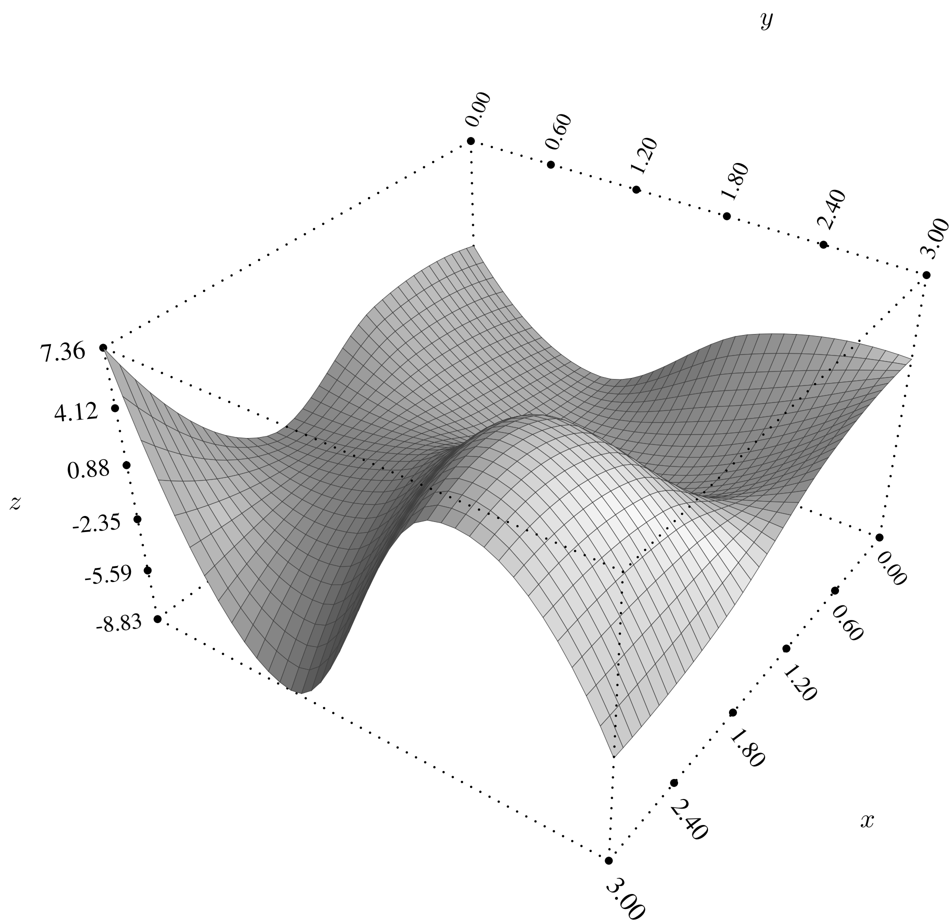


Figure 14.5: spline interpolation of Table 14.1

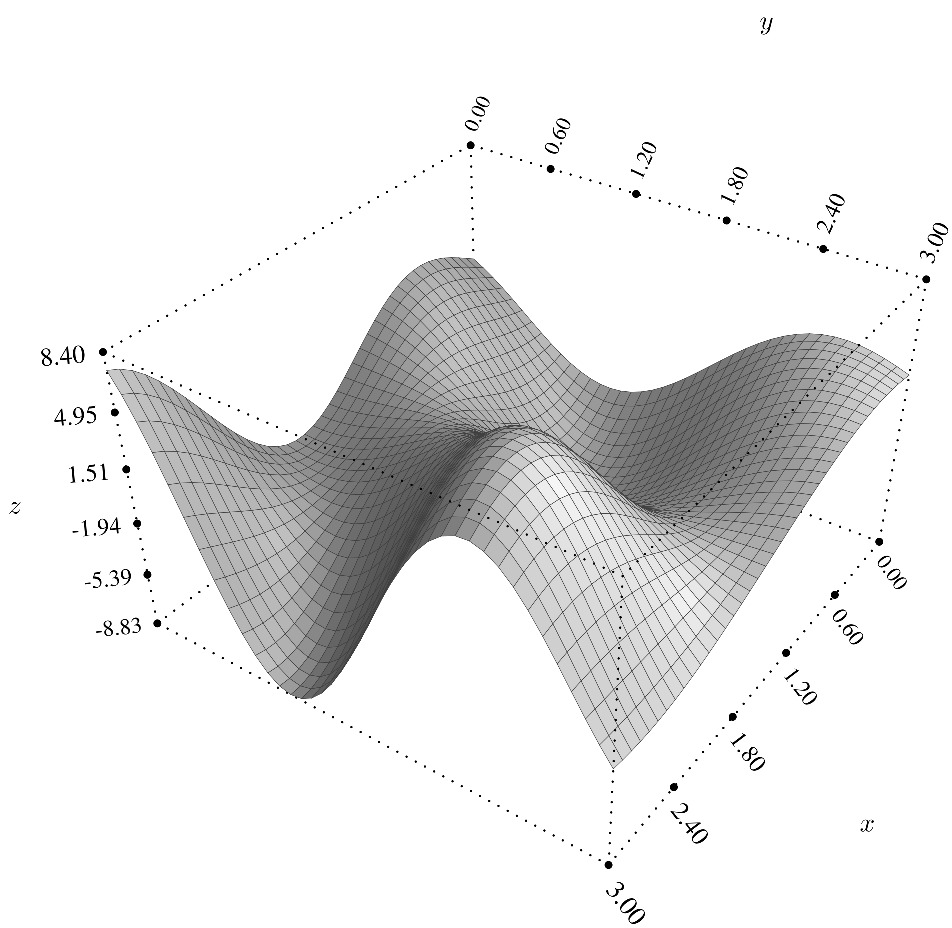


Figure 14.6: sinusoidal interpolation of Table 14.1

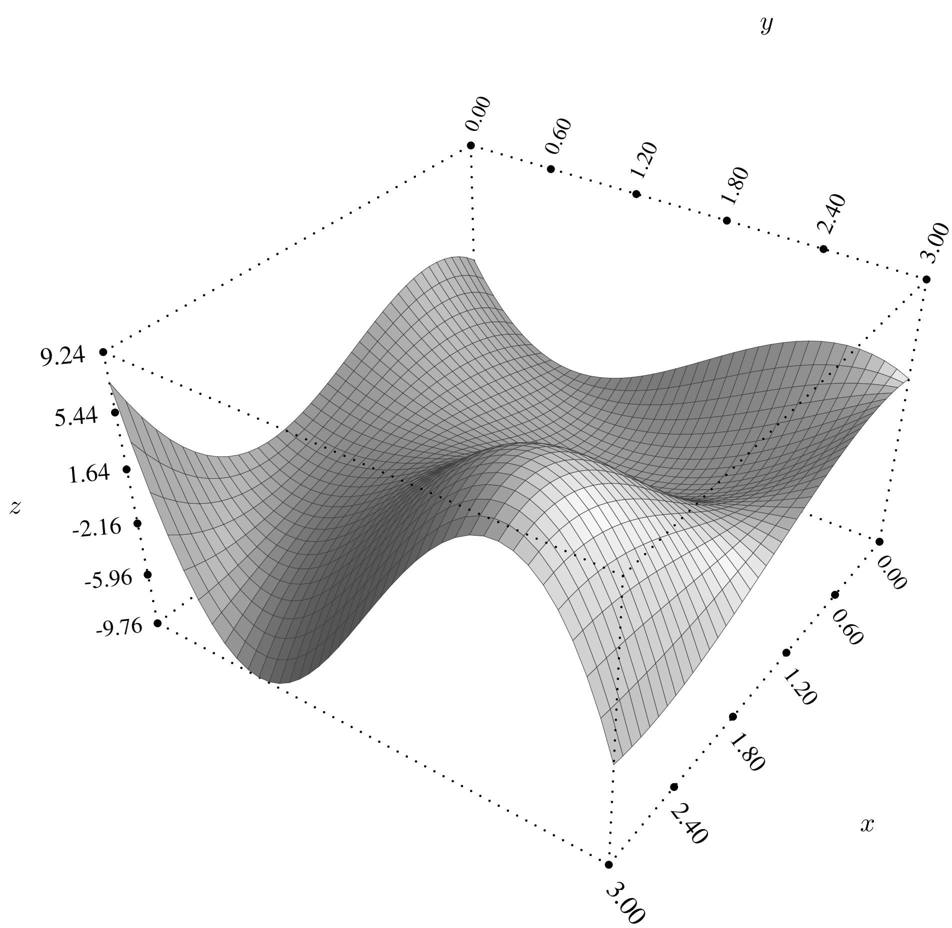


Figure 14.7: polynomial interpolation of Table 14.1

As you are undoubtedly gathering, the anomaly is systemic, creating fluctuations in even the most simplistic equations.

The Architect in *The Matrix Reloaded*

15 Continuous deformations

Several functions meant to expedite the task of mapping infinite continua to finite or semi-infinite subsets of themselves are provided by the `cop` library. Aside from general mathematical modelling applications, the main motivation for these functions is to adapt an unconstrained non-linear optimization solver such as `minpak` to constrained optimization problems by a change of variables.

The non-linear optimizers currently supported by virtual machine interfaces, `minpack` and `kinsol`, also allow a Jacobian matrix to be supplied by the user in either of two forms, which can be evaluated numerically by functions in this library.

15.1 Changes of variables

The functions documented in this section pertain to continuous maps of infinite intervals to finite or semi-infinite intervals.

half_line

This function takes a floating point number x and returns the number

$$\left(\frac{1+\tanh(x/k)}{2}\right)\sqrt{x^2+4}$$

where k is a fixed constant equal to 2.60080714.

The `half_line` function is plotted in Figure 15.1. Its purpose is to serve as a smooth map of the real line to the positive half line.

- Negative numbers are mapped to the interval $0\ldots1$.

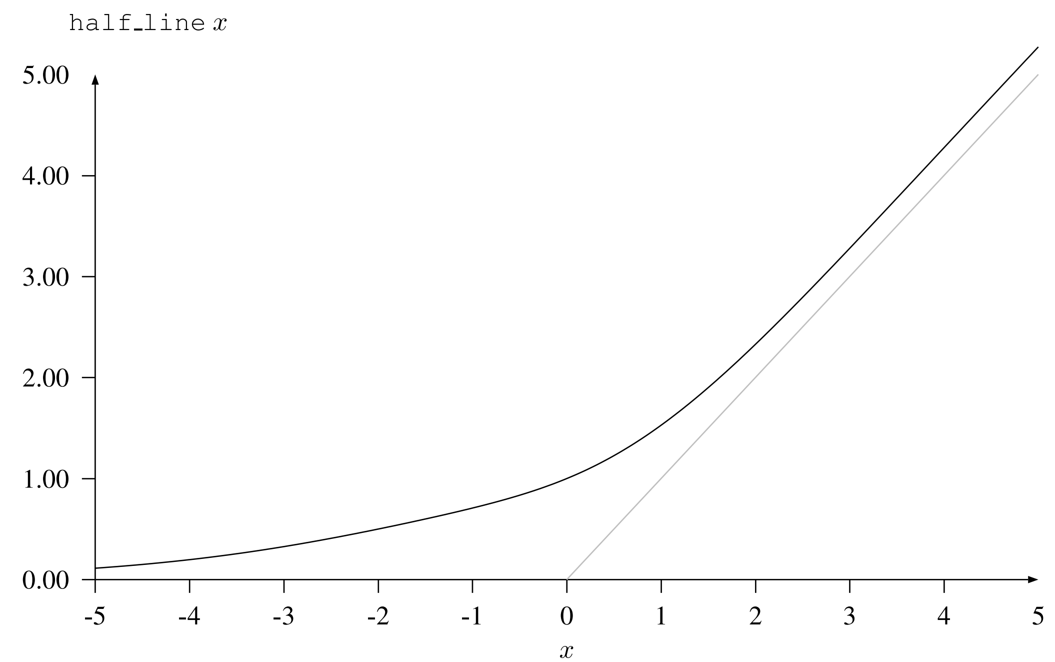


Figure 15.1: the `half_line` function maps the real line to the positive half line

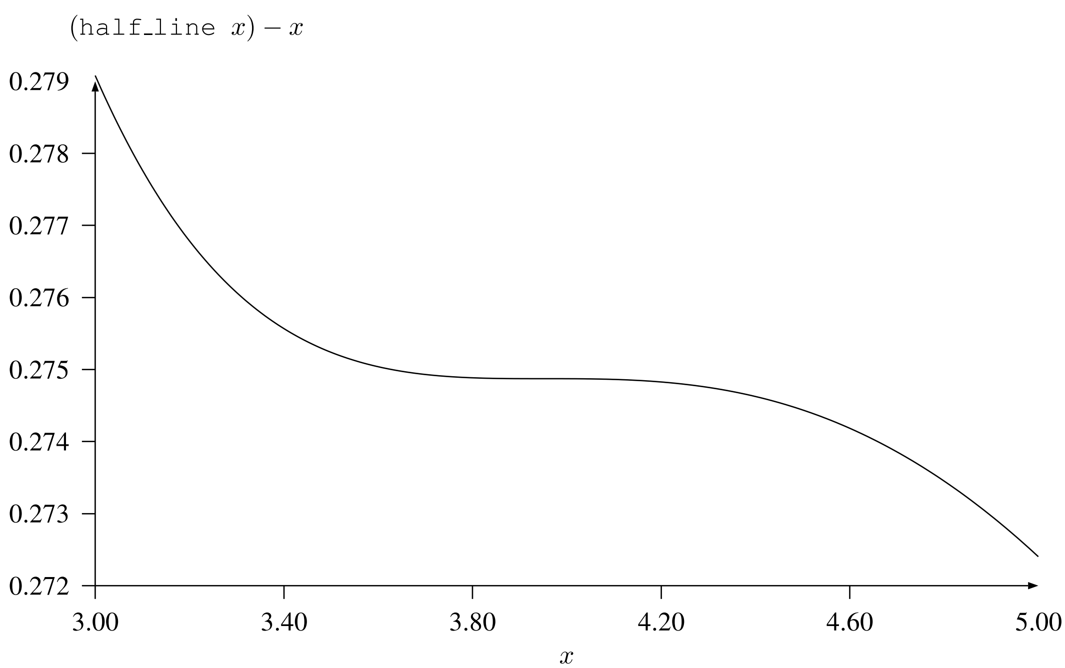


Figure 15.2: the `half_line` function converges monotonically on the positive side

- Positive numbers are mapped to the interval $1 \ldots \infty$.
- For large positive values of x, the function returns a value approximately equal to x.
- The constant k is chosen as the maximum value consistent with monotonic convergence from above, as shown in Figure 15.2.

The value of k is obtained by globally optimizing the function's first derivative subject to the constraint that it doesn't exceed 1.

over

Given a floating point number h, this function returns a function f that maps the real line to the interval $h \ldots \infty$ according to $f(x) = h + \texttt{half\_line}(x - h)$

under

Given a floating point number h, this function returns a function f that maps the real line to the interval $-\infty \ldots h$ according to $f(x) = h - \texttt{half\_line}(h - x)$.

Similarly to the `half_line` function, `over` h has a fixed point at infinity, whereas `under` h has a fixed point at negative infinity.

between

This function takes a pair of floating point numbers (a, b) with $a < b$ and returns a function f that maps the real line to the interval $a \ldots b$.

- If a and b are infinite, then f is the identity function.
- If a is infinite and b is finite, then $f = \texttt{under}\ b$.
- If a is finite and b is infinite, then $f = \texttt{over}\ a$.
- If a and b are both finite, then

$$f(x) = c + w \tanh \frac{x - c}{w}$$

where $c = (a + b)/2$ and $w = b - a$.

For the finite case, the function f has a fixed point and unit slope at $x = c$, the center of the interval.

chov

This function takes a list of pairs of floating point numbers $\langle (a_0, b_0) \ldots (a_n, b_n) \rangle$, and returns a function that maps a list of floating point numbers $\langle x_0 \ldots x_n \rangle$ to a list of floating point numbers $\langle y_0 \ldots y_n \rangle$ such that $y_i = (\texttt{between}\ (a_i, b_i))\ x_i$.

To solve a constrained non-linear optimization problem for a function $f : \mathtt{R}^n \rightarrow \mathtt{R}^m$ with initial guess $i \in \mathtt{R}^n$ and optimal output $o \in \mathtt{R}^m$ an expression of the form

$$x \texttt{ = (chov } c\texttt{) minpack..lmdir(}f\texttt{+ chov } c\texttt{,}i\texttt{,}o\texttt{)}$$

can be used, where $c = \langle (a_1, b_1) \dots (a_n, b_n) \rangle$ expresses constraints on each variable in the domain of f.

15.2 Partial differentiation

The functions documented in this section are suitable for obtaining partial derivatives of real valued functions of several variables.

jacobian

Given a pair of natural numbers (m, n), this function returns a function that takes a function $f : \mathtt{R}^n \rightarrow \mathtt{R}^m$ as an input, and returns a function $J : \mathtt{R}^n \rightarrow \mathtt{R}^{m \times n}$ as an output. The input to f and J is represented as a list $\langle x_1 \dots x_n \rangle$ of floating point numbers. The output from f is represented as a list of floating point numbers $\langle y_1 \dots y_m \rangle$, and the output from J as a list of lists of floating point numbers

$$\langle \langle d_{11} \dots d_{1n} \rangle \dots \langle d_{m1} \dots d_{mn} \rangle \rangle$$

For each i ranging from 1 to m, and for each j ranging from 1 to n, the value of d_{ij} is the incremental change observed in the value of y_i per unit of difference in x_j when f is applied to the argument $\langle x_1 \dots x_n \rangle$.

The Jacobian is customarily envisioned as a matrix of partial derivatives. If the function f is expressed in terms of an ensemble of m single valued functions of n variables,

$$f = \texttt{<.} f_1 \dots f_m \texttt{>}$$

then $J\langle x_1 \dots x_n \rangle$ contains entries d_{ij} given by

$$d_{ij} = \frac{\partial f_i}{\partial x_j} \langle x_1 \dots x_n \rangle$$

with these differences evaluated by the differentiation routines from the GNU Scientific Library. This representation of the Jacobian matrix is consistent with calling conventions used by the virtual machine's `kinsol` and `minpack` interfaces.

A simple example of the `jacobian` function is shown in Listing 15.1. When this source text is compiled, the following results are displayed.

```
$ fun flo cop jac.fun --show
<
   <1.000000e-00,1.000000e-00>,
   <0.000000e+00,-9.040721e-01>,
   <2.700000e+00,1.400000e+00>>
```

Listing 15.1 example of Jacobian function usage

```
#import std
#import nat
#import flo
#import cop

f = <.plus:-0.,sin+~&th,times+~&hthPX>

d = %eLLP (jacobian(3,2) f) <1.4,2.7>
```

A more complicated example of the `jacobian` function is shown in Listing 1.6 on page 33.

`jacobian_row`

Given a natural number n, this function constructs a function that takes a function $f : \mathtt{R}^n \rightarrow \mathtt{R}^m$ as an input, and returns a function $J : (\{0 \dots m-1\} \times \mathtt{R}^n) \rightarrow \mathtt{R}^n$ as an output.

- The input to f is represented as a list of floating point numbers $\langle x_1 \dots x_n \rangle$.
- The output from f is represented as a list of floating point numbers $\langle y_1 \dots y_m \rangle$.
- The input to J is represented as a pair $(i, \langle x_1 \dots x_n \rangle)$, where i is a natural number from 0 to $m-1$, and x_j is a floating point number.
- The output from J is represented as a list of floating point numbers $\langle d_1 \dots d_n \rangle$.

For each j ranging from 1 to n, the value of d_j is the incremental change observed in the value of y_{i+1} per unit of difference in x_j when f is applied to the argument $\langle x_1 \dots x_n \rangle$.

The purpose of the `jacobian_row` function is to allow an individual row of the Jacobian matrix to be computed without computing the whole matrix. The number i in the argument $(i, \langle x_1 \dots x_n \rangle)$ to the function (`jacobian_row` n) f is the row number, starting from zero. A definition of `jacobian` in terms of `jacobian_row` would be the following.

```
jacobian("m","n") "f" = (jacobian_row"n" "f")*+ iota"m"*-
```

Several functions in the `kinsol` and `minpack` library interfaces allow the Jacobian to be specified by a function with these calling conventions, so as to save time or memory in large optimization problems. Further details are documented in the `avram` reference manual.

Can you learn stuff that you haven't been programmed with, so you can be, you know, more human, and not such a dork all the time?

John Connor in *Terminator 2 – Judgment Day*

16
Linear programming

The `lin` library contains functions and data structures in support of linear programming problems. These features attempt to present a convenient, high level interface to the virtual machine's linear programming facilities, which are provided currently by the free third party libraries `glpk` and `lpsolve`. Enhancements to the basic interface include symbolic names for variables, positive and negative solutions, and costs proportional to magnitudes.

A few standard matrix operations are also included in this library as wrappers for the more frequently used virtual machine library functions, such as solutions of sparse systems and solutions in arbitrary precision arithmetic using the `mpfr` library.

Replacement functions implemented in virtual code are automatically invoked on platforms lacking interfaces to some of these libraries (`lapack`, `umf`, and `lpsolve` or `glpk`). These allow a nominal form of cross platform compatibility, but are not competitive in performance with native code implementations.

16.1 Matrix operations

The mathematical concept of an $n \times m$ matrix has a concrete representation as a list of lists of numbers, with one list for each row of the matrix as this diagram depicts.

$$\begin{pmatrix} a_{11} & \dots & a_{1m} \\ \vdots & \ddots & \vdots \\ a_{n1} & \dots & a_{nm} \end{pmatrix} \Leftrightarrow \begin{array}{l} \texttt{<} \\ \quad \texttt{<}a_{11} \dots a_{1m}\texttt{>,} \\ \quad \vdots \\ \quad \texttt{<}a_{n1} \dots a_{nm}\texttt{>>} \end{array}$$

This representation is assumed by the matrix operations documented in this section except as otherwise noted, and by the virtual machine model in general.

mmult

Given a pair of lists of lists of floating point numbers (a, b) representing matrices, this function returns a list of lists of floating point numbers representing their product, the matrix $c = ab$. For an $m \times n$ matrix a and an $n \times p$ matrix b, the product c is defined as then $m \times p$ matrix with

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj}$$

minverse

Given a list of lists of floating point numbers representing an $n \times n$ matrix a, this function returns a matrix b satisfying $ab = I$ if it exists, where I is the $n \times n$ identity matrix. If no such b exists, the result is unspecified. The identity matrix is defined as that which has $I_{ij} = 1$ for i equal to j, and zero otherwise.

Computing the inverse of a matrix may be of pedagogical interest but is less efficient for solving systems of equations than the following function. This rule of thumb applies even if a given matrix needs to be solved with many different vectors, and even if the inverse can be computed at no cost (i.e., off line in advance).

msolve

Given a pair (a, b) representing an $n \times n$ matrix and an $n \times 1$ matrix of floating point numbers, respectively, this function returns a representation of an $n \times 1$ matrix x satisfying $ax = b$. Contrary to the usual representation of matrices as lists of lists, this function represents b and x as lists $\langle b_{11} \dots b_{n1} \rangle$ and $\langle x_{11} \dots x_{n1} \rangle$.

The `msolve` function calls the corresponding `lapack` routine if available, but otherwise solves the system in virtual code using a Gauss-Jordan elimination procedure with pivoting.

mp_solve

This function has the same calling conventions as `msolve`, but uses arbitrary precision numbers in `mpfr` format (type `%E`).

sparso

This function solves the matrix equation $ax = b$ for x given the pair (a, b) where a has a sparse matrix representation, and x and b are represented as lists $\langle x_{11} \dots x_{n1} \rangle$ and $\langle b_{11} \dots b_{n1} \rangle$. The sparse matrix representation is the list of tuples $((i-1, j-1), a_{ij})$ wherein only the non-zero values of a_{ij} are given, and i and j are natural numbers.

mp_sparso

This function has the same calling conventions as sparso but solves systems using arbitrary precision numbers in mpfr format.

The sparso function will use the umf library for solving sparse systems efficiently if the virtual machine is configured with an interface to it. If not, the system is converted to the dense representation and solved by msolve. There is no native code sparse matrix solver for mpfr numbers, so mp_sparso always converts its input to dense matrix representations and solves it by mp_solve.

16.2 Continuous linear programming

There are two linear programming solvers in this library, with one closely following the calling convention of the virtual machine interfaces to glpk and lpsolve, and the other allowing a higher level, symbolic specification of the problem. The latter employs a record data structure as documented below.

16.2.1 Data structures

The linear programming problem in standard form is that of finding an $n \times 1$ matrix X to minimize a cost CX for a known $1 \times n$ matrix C, subject to the constraints that $AX = B$ for given matrices A and B, and all $X_{i1} \geq 0$.

Letting $x_i = X_{i1}$, $b_i = B_{i1}$, $c_i = C_{1i}$, and $z = \sum_{i=1}^{n} c_i x_i$ the constraint $AX = B$ is equivalent to a system of linear equations.

$$\sum_{j=1}^{n} A_{ij} x_j = b_i$$

In practice, most A_{ij} values are zero. A more user-friendly formulation of this problem than the standard form would admit the following features.

- constraints on the variables x_i having arbitrary upper and lower bounds

$$l_i \leq x_i \leq u_i$$

- costs allowed to depend on magnitudes

$$z + \sum_{i=1}^{n} t_i |x_i|$$

- an assignment of symbolic names to x values $\langle s_1 : x_1, \ldots s_n : x_n \rangle$
- the system of equations encoded as a list of pairs of the form $(\langle (A_{ij}, s_j) \ldots \rangle, b_i)$ with only the non-zero coefficients A_{ij} enumerated

A record data structure is used to encode the problem specification in the latter form, making it suitable for automatic conversion to the standard form.

linear_system

This function is the mnemonic for a record having the following field identifiers, which specifies a linear programming problem in terms of the notation introduced above, with numeric values represented as floating point numbers and s_i values as character strings.

- `lower_bounds` – the set of assignments $\{s_1 : l_1 \ldots s_n : l_n\}$
- `upper_bounds` – the set of assignments $\{s_1 : u_1 \ldots s_n : u_n\}$
- `costs` – the set of assignments $\{s_1 : c_1 \ldots s_n : c_n\}$
- `taxes` – the set of assignments $\{s_1 : t_1 \ldots s_n : t_n\}$
- `equations` – the set $\{(\{(A_{ij}, s_j) \ldots\}, b_i) \ldots\}$
- `derivations` – a field used internally by the library

The members of these sets may of course be given in any order. Any unspecified bounds are treated as unconstrained. All costs must be specified but taxes are optional.

For performance reasons, this record structure performs no validation or automatic initialization, so the user is required to construct it consistently.

16.2.2 Functions

The following functions are used in solving linear programming problems.

standard_form

This function takes a record of type `_linear_system` and transforms it to the standard from by defining supplementary variables and equations as needed.

- All `lower_bounds` are transformed to zero.
- All `upper_bounds` are transformed to infinity.
- The `taxes` are transformed to `costs`.

Information allowing a solution of the original specification to be inferred from a solution of the transformed system is stored in the `derivations` field.

The `standard_form` function doesn't need to be used explicitly unless these transformations are of some independent interest, because it is invoked automatically by the next function.

solution

Given a record of type `_linear_system` specifying a linear programming problem, this function returns a list of assignments $\langle s_i : x_i, \dots \rangle$, where each s_i is a symbolic name for a variable obtained from the `equations` field, and x_i is a floating point number giving the optimum value of the variable. Variables equal to zero are omitted. If no feasible solution exists, the empty list is returned.

lp_solver

This function solves linear programming problems by a low level, high performance interface. The input to the function is a linear programming problem specified by a triple

$$(\langle c_1 \dots c_n \rangle, \langle ((i-1, j-1), A_{ij}) \dots \rangle, \langle b_1 \dots b_m \rangle)$$

where c_i and b_i are as documented in Section 16.2.1, and the remaining parameter is the sparse matrix representation of the constraint matrix A as explained in relation to the `sparso` function on page 358. The result is a list of pairs $\langle (i-1, x_i) \dots \rangle$, giving the optimum value of each non-zero variable with its index numbered from zero as a natural number. If no feasible solution exists, the empty list is returned.

The `lp_solver` function is called by the `solution` function, and it calls one of the `glpk` or `lpsolve` functions to do the real work. If the virtual machine is not configured with interfaces to these libraries, it falls through to this replacement function.

replacement_lp_solver

This function has identical semantics and calling conventions to the `lp_solver` function documented above.

The replacement function is implemented purely in virtual code without calling `lpsolve` or `glpk` and can serve as a correct reference implementation of a linear programming solver for testing purposes, but it is too slow for production use, mainly because it exhaustively samples every vertex of the convex hull.

16.3 Integer programming

Integer programming problems are an additionally constrained form of linear programming problems in which the solutions x_i are required to take integer values. If some but not all x_i are required to be integers, then the problem is called a mixed integer programming problem.

Current versions of the virtual machine can be configured with an interface to the `lpsolve` library providing for the solution of integer and mixed integer programming problems, and this capability is accessible in Ursala by way of the `lin` library.[1] An integer programming problem is indicated by setting either or both of these to additional fields in the `linear_system` data structure.

- `integers` – an optional set of symbolic names $\{s_i \dots s_j\}$ identifying the integer variables
- `binaries` – an optional set of symbolic names $\{s_i \dots s_j\}$ identifying the binary variables

The binary variables not only are integers but are constrained to take values of 0 or 1. These sets must be subsets of the names of variables appearing in the `equations` field. A data structure with these fields initialized may be passed to the `solution` function as usual, and the solution, if found, will meet these constraints although it will still use the floating point numeric representation. Solution of an integer programming problem is considerably more time consuming than a comparable continuous case.

There is no replacement function for mixed integer programming problems, but there is a lower level, higher performance interface suitable for applications in which the the standard form of the system is known.

mip_solver

This function solves linear programming problems given a linear system as input in the form

$$((\langle bv_k \dots \rangle, \langle iv_k \dots \rangle), \langle c_1 \dots c_n \rangle, \langle ((i-1, j-1), A_{ij}) \dots \rangle, \langle b_1 \dots b_m \rangle)$$

where natural numbers bv_k are indices of binary variables, iv_k are indices of integer variables, c_i and b_i are as documented in Section 16.2.1, and the remaining parameter is the sparse matrix representation of the constraint matrix A as explained in relation to the `sparso` function on page 358. The result is a list of pairs $\langle (i-1, x_i) \dots \rangle$, giving the optimum value of each non-zero variable with its index numbered from zero as a natural number. If no feasible solution exists, the empty list is returned.

[1]The integer programming interface to `lpsolve` was introduced in Avram version 0.12.0, and remains backward compatible with earlier code. The features described in this section were introduced in Ursala version 0.7.0.

I don't set a fancy table, but my kitchen's awful homey.
Anthony Perkins in *Psycho*

17
Tables

This chapter documents a small selection of functions intended to facilitate the construction of tables of numerical data with publication quality typesetting. These functions are particularly useful for tables with hierarchical headings that might be more difficult to typeset manually, and for tables whose contents come from the output of an application developed in Ursala.

The tables are generated as LaTeX code fragments meant to be included in a document or presentation. They require the document that includes them to use the LaTeX `booktabs` package. The functions are defined in the `tbl` library.

17.1 Short tables

A table is viewed as having two parts, which are the headings and the body.

- The body is a list of columns, wherein each column is either a list of character strings or a list of floating point numbers.
- The headings are a list of trees of lists of strings (type `%sLTL`).
  - Each non-terminal node in a tree is a collective heading for the subheadings below it.
  - Each terminal node is a heading for an individual column.
  - The total number of terminal nodes in the list of trees is equal to the number of columns.

The character strings in the table headings or columns can contain any valid LaTeX code. Its validity is the user's responsibility.

table

This function takes a natural number n as an argument, and returns a function that generates LaTeX code for a `tabular` environment from an input (h, b) of type `%sLTLeLsLULX` containing headings h and a body b as described above. Any columns in the body containing floating point numbers are typeset in fixed decimal format with n decimal places.

A simple but complete example of a table constructed by this function is shown in Listing 17.1. In practice, the table contents are more likely to be generated algorithmically than written manually in the source text, as the argument to the `table` function can be any expression evaluated at compile time. The example is otherwise realistic insofar as it demonstrates the typical way in which a table is written to a file by the `#output dot'tex'` directive with the identity function as a formatter. An alternative would be the usage

```
#output dot'tex' table3

atable = (headings,body)
```

with further variations possible. In any case, the table may then be incorporated into a document by a code fragment such as the following.

```
\usepackage{booktabs}
\begin{document}
...
\begin{table}
\begin{center}
\input{atable}
\end{center}
\caption{the tables are turning}
\label{alabel}
\end{table}
```

This code fragment is based on the assumption that the user intends to have the table centered in a floating table environment, with a caption and label, but these choices are all at the user's option. Only the actual `tabular` environment is stored in the file. Also note that the file name is the same as the identifier used in the source with the `.tex` suffix appended, but the suffix is implicit in the LaTeX code. See Section 7.4.4 on page 264 for more information about the `#output` directive.

The result from Listing 17.1 is shown in Table 17.1. As the example shows, headings with multiple strings are typeset on multiple lines, all headings are vertically centered, and all columns are right justified.

A more complicated example of table heading specifications is shown on page 49 and the result displayed in Table 1.1. These headings are generated algorithmically by the user application in Listing 1.11.

Listing 17.1 simple example of the `table` function usage

```
#import std
#import nat
#import tbl

headings = # a list of trees of lists of strings

<
   <'name'>^: <>,    # table heading
   <'foo'>^: <
      <'bar','baz'>^: <>,   # subheadings
      <'rank'>^: <>>>

body = # list of lists of either strings or numbers

<
   <'x','y','z'>,  # each list is a column
   <1.,2.,3.>,
   <4.,5.,6.>>

#output dot'tex' ~&

atable = table3(headings,body)
```

	foo	
name	bar baz	rank
x	1.000	4.000
y	2.000	5.000
z	3.000	6.000

Table 17.1: table generated by Listing 17.1

Listing 17.2 usage of the `sectioned_table` function

```
#import std
#import nat
#import tbl

headings = # a list of trees of lists of strings

<
   <'name'>^: <>,
   <'foo'>^: <<'bar','baz'>^: <>,<'rank'>^: <>>>

body = # a list of lists of columns

<
   <<'u','v','w'>,<7.,8.,9.>,<0.,1.,2.>>,
   <<'x','y','z'>,<1.,2.,3.>,<4.,5.,6.>>>

#output dot'tex' ~&

setab = sectioned_table3(headings,body)
```

sectioned_table

This function takes a natural number n to a function that takes a pair (h, b) to a LaTeX code fragment for a table with headings h and body b. The body b is a list of lists of columns (type `%eLsLULL`) with each list of columns to be typeset in a separate section delimited by horizontal rules. Floating point numbers in the body are typeset in fixed decimal format with n places.

Note that although the same headings can be used for a sectioned table as for a table, the body of the latter is of a different type. An example of the `sectioned_table` function is shown in Listing 17.2, and the table it generates is shown in Table 17.2, with horizontal rules serving to separate the table sections.

There is no automatic provision for vertical rules, because the author of the LaTeX `booktabs` package considers vertical rules bad typographic design in tables, but users may elect to customize the output table manually or by any post processor of their design.

17.2 Long tables

A couple of functions documented in this section are useful for constructing tables that are too long to fit on a page. These require the document that includes them to use the LaTeX `longtable` package.

The general approach is to construct tables normally by one of the functions described previously (`table` or `sectioned_table`), and then to transform the result to a long

	foo	
name	bar baz	rank
u	7.000	0.000
v	8.000	1.000
w	9.000	2.000
x	1.000	4.000
y	2.000	5.000
z	3.000	6.000

Table 17.2: the table generated by Listing 17.2

table format by way of a post processing operation. The `longtable` environment combines aspects of the ordinary `table` and `tabular` environments, precluding postponement of the choice of a caption and label as in previous examples, and hence requiring calling conventions such as the following.

`elongation`

Given a character string containing LaTeX code specifying a title, this function returns a function that transforms a given `tabular` environment in a list of strings to the corresponding `longtable` environment having that title.

A typical usage of this function would be in an expression of the form

$$\texttt{elongation}\langle \textit{title}\rangle\ \ (\texttt{[sectioned\_]table}\ n)\ \ (\langle \textit{headings}\rangle, \langle \textit{body}\rangle)$$

`label`

Given a character string specifying a label, this function returns a function that transforms a given `longtable` environment in a list of strings to a `longtable` environment having that label.

A typical usage of this function would be in an expression of the form

$$\texttt{label}\langle \textit{name}\rangle\ \ \texttt{elongation}\langle \textit{title}\rangle\ \ (\texttt{[sectioned\_]table}\ n)\ (\langle \textit{headings}\rangle, \langle \textit{body}\rangle)$$

The table thus obtained can be cross referenced in the document by the usual LaTeX label features such as `\ref{`⟨*name*⟩`}` and `\pageref{`⟨*name*⟩`}`.

17.3 Utilities

A further couple of functions described in this section may be helpful in preparing the contents of a table.

Listing 17.3 some uses of the `vwrap` function

```
#import std
#import nat
#import tbl

#output dot'tex' table0

chab = # ISO codes for upper and lower case letters

vwrap5(
   ~&iNCNVS <'letter','code'>,
   <.~&rNCS,~&hS+ %nP*+ ~&lS> ~&riK10\letters num characters)

pows = # first seven powers of numbers 1 to 7

vwrap7(
   ~&iNCNVS <'$n$','$m$','$n^m$'>,
   ~&hSS %nP** <.~&lS,~&rS,power*> ~&ttK0 iota 8)
```

letter	code	letter	code	letter	code	letter	code	letter	code
A	65	L	76	W	87	h	104	s	115
B	66	M	77	X	88	i	105	t	116
C	67	N	78	Y	89	j	106	u	117
D	68	O	79	Z	90	k	107	v	118
E	69	P	80	a	97	l	108	w	119
F	70	Q	81	b	98	m	109	x	120
G	71	R	82	c	99	n	110	y	121
H	72	S	83	d	100	o	111	z	122
I	73	T	84	e	101	p	112		
J	74	U	85	f	102	q	113		
K	75	V	86	g	103	r	114		

Table 17.3: character table generated by Listing 17.3

n	m	n^m	n	n^m	n	n^m	n	n^m	n	n^m	n	n^m	n	n^m
1	1	1	2	2	3	3	4	4	5	5	6	6	7	7
1	2	1	2	4	3	9	4	16	5	25	6	36	7	49
1	3	1	2	8	3	27	4	64	5	125	6	216	7	343
1	4	1	2	16	3	81	4	256	5	625	6	1296	7	2401
1	5	1	2	32	3	243	4	1024	5	3125	6	7776	7	16807
1	6	1	2	64	3	729	4	4096	5	15625	6	46656	7	117649
1	7	1	2	128	3	2187	4	16384	5	78125	6	279936	7	823543

Table 17.4: table of powers generated by Listing 17.3

vwrap

> This function takes a natural number n as an argument, and returns a function that transforms the headings and body of a table given as a pair (h, b) of type `%sLTLeLsLULX` to a result of the same type. The transformation partitions the columns vertically into n approximately equal parts and places them side by side, with the headings adjusted accordingly. Repeated columns in the result are deleted.

If a table is narrow enough that most of the space beside it on a page is wasted, the `vwrap` function allows a more space efficient alternative layout to be generated with no manual revisions to the heading and column specifications required.

Two examples of the `vwrap` function are shown in Listing 17.3, with the resulting tables displayed in Table 17.3 and Table 17.4. Without the `vwrap` function, both tables would have only two or three narrow columns and be too long to fit on the page.

Table 17.4 demonstrates the effect of deleting repeated columns by the `vwrap` function. Because the same values of m are applicable across the table, the column for m is displayed only once. A table made from the original body in Listing 17.3 would have included the repeated m values.

scientific_notation

> This function takes a character string as an argument and detects whether it is a syntactically valid decimal number in exponential notation. If not, the argument is returned as the result. In the alternative, the result is a LaTeX code fragment to typeset the number as a product of the mantissa and a power of ten.

This function can be demonstrated as follows.

```
$ fun tbl --m="scientific_notation '6.022e+23'" --c %s
'6.022$\times 10^{23}$'
```

The result appears as $6.022{\times}10^{23}$ in a typeset document.

The `scientific_notation` function need not be invoked explicitly to get this effect in a table, because it applies automatically to any column whose entries are character strings in exponential format. Floating point numbers can be converted to strings in exponential format by the `printf` function as explained in Section 13.9.

The core network of the grid must be accessed.

The Keymaker in *The Matrix Reloaded*

18

Lattices

Data of type t%G, using the grid type constructor explained in Chapter 3, are supported by a variety of operations defined in the lat library and documented in this chapter. These include basic construction and deconstruction functions, iterators analogous to some of the usual operations on lists, and higher order functions implementing the induction patterns that are the main reason for using lattices.

18.1 Constructors

The first thing necessary for using a lattice is to construct one, which can be done easily by the grid function.

grid

This function takes a pair with a list of lists of vertices on the left and a list of adjacency relations on the right, $(\langle\langle v_{00} \dots v_{0n_0}\rangle \dots \langle v_{m0} \dots v_{mn_m}\rangle\rangle, \langle e_0 \dots e_{m-1}\rangle)$. It returns a lattice populated by the vertices and connected according to the adjacency relations.

- The i-th adjacency relation e_i is a function taking pairs of vertices $(v_{ij}, v_{i+1,k})$ as input, with the left vertex from the i-th list and the right vertex from the succeeding one.
- A connection is made between any pair of vertices $(v_{ij}, v_{i+1,k})$ for which the corresponding relation e_i returns a non-empty value.
- Any vertex not reachable by some sequence of connections originating from at least one vertex v_{0j} in the first list is omitted from the output lattice.

The `grid` function allows the input list of adjacency relations to be truncated if subsequent relations are the same as the last one in the list.

A few small examples of lattices constructed by this function should clarify the description. In these examples, the verticies are the characters `` `a ``, `` `b ``, `` `c `` and `` `d ``, expressed in strings rather than lists for brevity. The first example shows a fully connected lattice, which is obtained by using a (truncated) list of adjacency relations that are always true.[1]

```
$ fun lat --m="grid/<'a','ab','abc','abcd'> <&!>" --c %cG
<
   [0:0: `a^: <1:0,1:1>],
   [
      1:1: `b^: <2:0,2:1,2:2>,
      1:0: `a^: <2:0,2:1,2:2>],
   [
      2:2: `c^: <2:0,2:1,2:2,2:3>,
      2:1: `b^: <2:0,2:1,2:2,2:3>,
      2:0: `a^: <2:0,2:1,2:2,2:3>],
   [
      2:3: `d^: <>,
      2:2: `c^: <>,
      2:1: `b^: <>,
      2:0: `a^: <>]>
```

This example shows a lattice with each letter connected only to those that don't precede it in the alphabet.

```
$ fun lat --m="grid/<'a','ab','abc','abcd'> <lleq>" --c %cG
<
   [0:0: `a^: <1:0,1:1>],
   [
      1:1: `b^: <2:1,2:2>,
      1:0: `a^: <2:0,2:1,2:2>],
   [
      2:2: `c^: <2:2,2:3>,
      2:1: `b^: <2:1,2:2,2:3>,
      2:0: `a^: <2:0,2:1,2:2,2:3>],
   [
      2:3: `d^: <>,
      2:2: `c^: <>,
      2:1: `b^: <>,
      2:0: `a^: <>]>
```

The next example shows the degenerate case of a lattice obtained by using equality as the adjacency relation, resulting in most letters being unreacheable and therefore omitted.

[1] Remember to execute `set +H` before trying this example to suppress interpretation of the exclamation point by the shell.

```
$ fun lat --m="grid/<'a','ab','abc','abcd'> <==>" --c %cG
<
   [0:0: `a^: <0:0>],
   [0:0: `a^: <0:0>],
   [0:0: `a^: <0:0>],
   [0:0: `a^: <>]>
```

Finally, we have an example of a lattice generated with a branching pattern chosen at random. Each vertex has a 50% probability of being connected to each vertex in the next level.

```
$ fun lat --m="grid/<'a','ab','abc','abcd'> <50%~>" --c %cG
<
   [0:0: `a^: <1:0,1:1>],
   [1:1: `b^: <1:0,1:1>,1:0: `a^: <1:0>],
   [1:1: `c^: <2:1,2:2>,1:0: `a^: <2:0>],
   [2:2: `d^: <>,2:1: `c^: <>,2:0: `b^: <>]>
```

Along with constructing a lattice goes the need to deconstruct one in order to access its components. Several functions for this purpose follow.

levels

Given a lattice of the form `grid(<`v_{00}`>:`v`,`e`)`, (i.e., with a unique root vertex v_{00}) this function returns the list of lists of vertices `<`v_{00}`>:`v, subject to the removal of unreachable vertices.

lnodes

This function is equivalent to `~&L+ levels`, and useful for making a list of the nodes in a lattice without regard for their levels.

These functions can be demonstrated as follows.

```
$ fun lat --m="levels grid/<'a','ab','abc'> <&!>" --c %sL
<'a','ab','abc'>
$ fun lat --m="lnodes grid/<'a','ab','abc'> <&!>" --c %s
'aababc'
```

A unique root vertex is a needed for these algorithms, but this restriction is not severe in practice because a root normally can be attached to a lattice if necessary.

edges

Given a lattice with a unique root vertex, this function returns the list of lists of addresses for the vertices by levels.

This function may be useful in user-defined *ad hoc* lattice deconstruction functions. Here is an example.

```
$ fun lat --m="edges grid/<'a','ab','abc'> <&!>" --c %aLL
<<0:0>,<1:0,1:1>,<2:0,2:1,2:2>>
```

sever

Given a lattice of type t%G, with a unique root vertex, this function returns a lattice of type t%GG by substituting each vertex v with the sub-lattice containing only the vertices reachable from v, while preserving their adjacency relation.

The following example demonstrates this function.

```
$ fun lat --m="sever grid/<'a','ab','abc'> <&!>" --c %cGG
<
   [
      0:0: ^:<1:0,1:1> <
         [0:0: `a^: <1:0,1:1>],
         [
            1:1: `b^: <2:0,2:1,2:2>,
            1:0: `a^: <2:0,2:1,2:2>],
         [2:2: `c^: <>,2:1: `b^: <>,2:0: `a^: <>]>],
   [
      1:1: ^:<2:0,2:1,2:2> <
         [0:0: `b^: <2:0,2:1,2:2>],
         [2:2: `c^: <>,2:1: `b^: <>,2:0: `a^: <>]>,
      1:0: ^:<2:0,2:1,2:2> <
         [0:0: `a^: <2:0,2:1,2:2>],
         [2:2: `c^: <>,2:1: `b^: <>,2:0: `a^: <>]>],
   [
      2:2: (<[0:0: `c^: <>]>)^: <>,
      2:1: (<[0:0: `b^: <>]>)^: <>,
      2:0: (<[0:0: `a^: <>]>)^: <>]>
```

18.2 Combinators

The functions documented in this section are analogues to functions and combinators normally associated with lists, such as maps, folds, zips, and distributions. All of them require lattices with a unique root vertex.

ldis

Given a pair (x, g) where g is a lattice, this function returns a lattice derived from g by substituting each vertex v in g with the pair (x, v).

This function is analogous to distribution on lists, and can be demonstrated as follows.

```
$ fun lat -m="ldis/1 grid/<'a','ab','abc'> <&!>" -c %ncXG
<
   [0:0: (1,`a)^: <1:0,1:1>],
   [
      1:1: (1,`b)^: <2:0,2:1,2:2>,
      1:0: (1,`a)^: <2:0,2:1,2:2>],
   [
      2:2: (1,`c)^: <>,
      2:1: (1,`b)^: <>,
      2:0: (1,`a)^: <>]>
```

ldiz

This function takes a pair (x, g) where g is a lattice having a unique root vertex and x is a list having a length equal to the number of levels in g. The returned value is a lattice derived from g by substituting each vertex v on the i-th level with the pair (x_i, v), where x_i is the i-th item of x.

A simple demonstration of this function is the following.

```
$ fun lat --m="ldiz/'xy' grid/<'a','ab'> <&!>" --c %cWG
<
   [0:0: (`x,`a)^: <1:0,1:1>],
   [1:1: (`y,`b)^: <>,1:0: (`y,`a)^: <>]>
```

lmap

Given a function f, this function returns a function that takes a lattice g as input, and returns a lattice derived from g by substituting every vertex v in g with $f(v)$.

The `lmap` combinator on lattices is analogous to the `map` combinator on lists. This example shows the `lmap` of a function that duplicates its argument.

```
$ fun lat --m="(lmap ~&iiX) grid/<'a','ab'> <&!>" --c %cWG
<
   [0:0: (`a,`a)^: <1:0,1:1>],
   [1:1: (`b,`b)^: <>,1:0: (`a,`a)^: <>]>
```

lzip

Given a pair of lattices (a, b) with unique roots and identical branching patterns, this function returns a lattice c in which every vertex v is the pair (u, w) with u being the vertex at the corresponding position in a and w being the vertex at the corresponding position in b.

This function is comparable the the `zip` function on lists. The following example shows a lattice zipped to a copy of itself.

```
$ fun lat --m="lzip (~&iiX grid/<'a','ab'> <&!>)" --c %cWG
<
   [0:0: (`a,`a)^: <1:0,1:1>],
   [1:1: (`b,`b)^: <>,1:0: (`a,`a)^: <>]>
```

This operation has the same effect as the previous example, because `lmap ~&iiX` is equivalent to `lzip+ ~&iiX`.

lfold

Given a function f, this function constructs a function that traverses a lattice backwards toward the root, evaluating f at each vertex v by applying it to the pair $(v, \langle y_0 \dots y_n \rangle)$, where the y values are the outputs from f obtained previously when visiting the descendents of v. The overall result is that which is obtained when visitng the root.

The `lfold` combinator is analogous to the tree folding operator `^*` explained in Section 6.8.2 on page 219, but it operates on lattices rather than trees. The following simple example shows how the `lfold` combinator of the tree constructor converts a lattice into an ordinary tree (with an exponential increase in the number of vertices).

```
$ fun lat --m="lfold(^:) grid/<'a','ab','abc'> <&!>" -c %cT
`a^: <
   `a^: <`a^: <>,`b^: <>,`c^: <>>,
   `b^: <`a^: <>,`b^: <>,`c^: <>>>
```

A more practical example of the `lfold` combinator is shown in Listing 1.5 with some commentary on page 32.

18.3 Induction patterns

The benefit of working with a lattice is in effecting a computation by way of one or more of the transformations documented in this section. These allow an efficient, systematic pattern of traversal through a lattice, visiting a user defined function on each vertex, and allowing it to depend on the results obtained from neighboring vertices. Directions of traversal can be forward, backward, sideways, or a combination. These operations are also composable because the inputs and outputs are lattices in all cases.

Many of the algorithms concerning lattices have analogous tree traversal algorithms. As the previous example demonstrates, a lattice of type t`%G` can be converted to a tree of type t`%T` without any loss of information, and operating on the tree would be more convenient if it were not exponentially more expensive, because the tree is a simpler and more abstract representation. The combinators documented in this section therefore attempt to present

Listing 18.1 lattice transformation examples

```
#import std
#import nat
#import lat

x = grid/<'a','bc','def','ghij'> <&!>

xpress  = bwi :^/~&l ~&rdS; ~&i&& :/`(+ --')'+ mat`,
paths   = fwi ^rlrDlShiX2lNXQ\~&rv ~&l?\~&rdNCNC ~&rdPlLPDrlNCTS
roll    = swi ^H\~&r -$+ ~&lizyCX

neighbors =

fswi ^\~&rdvDlS :^/~&ll ^T(
   ~&lrNCC+ ~&rilK16rSPirK16lSPXNNXQ+ ~&rdPlrytp2X,
   ~&rvdSNC)
```

an interface to the user application whereby the lattice appears as a tree as far as possible. In particular, it is never necessary for the application to be concerned explicitly with the address fields in a lattice.

bwi

A function of the form `bwi` f maps a lattice x of type t`%G` to an isomorphic lattice y of type u`%G`. Each vertex w in y is given by $f(v, \langle z_0 \dots z_n\rangle)$, where v is the corresponding vertex in x and the z values are trees (of type u`%T`) populated by previous applications of f for the vertices reachable from v. The root of z_k is the value of f computed for the k-th neighboring vertex referenced by the adjacency list of v.

The `bwi` function is mnemonic for "backward induction", because the vertices most distant from the root are visited first. In this regard it is similar to the `lfold` function, but the argument f follows a different calling convention allowing it direct access to all relevant previously computed results rather than just those associated with the top level of descendents. The precise relationship between these two operations is summarized by the following equivalence.

`(bwi` f`)` x $\equiv$ `(lmap ~&l+ lfold ^\~&v` f`) sever` x

However, it would be very inefficient to implement the `bwi` function this way.

An example of backward induction is shown in the `xpress` function in Listing 18.1. This function is purely for illustrative purposes, attempting to depict the chain of functional dependence of each level on the succeeding ones in a backward induction algorithm. The argument to the `bwi` combinator is the function

`:^/~&l ~&rdS; ~&i&& :/`(+ --')'+ mat`,`

which is designed to operate on an argument of the form $(v, \langle z_0 \dots z_n\rangle)$, for a character

v and a list of trees of strings z_i. It returns a single character string by flattening and parenthesizing the roots of the trees and inserting the character v at the head. The subtrees of z_i are ignored. With Listing 18.1 stored in a file named `lax.fun`, this function can be demonstrated as follows.

```
$ fun lat lax -m="xpress grid/<'a','bc','def'> <&!>" -c %sG
<
   [0:0: 'a(b(d,e,f),c(d,e,f))'^: <1:0,1:1>],
   [
      1:1: 'c(d,e,f)'^: <2:0,2:1,2:2>,
      1:0: 'b(d,e,f)'^: <2:0,2:1,2:2>],
   [2:2: 'f'^: <>,2:1: 'e'^: <>,2:0: 'd'^: <>]>
```

fwi

A function of the form `fwi` f transforms a lattice x of type t`%G` to an isomorphic lattice y of type u`%G`. To compute y, the lattice x is traversed beginning at the root.

- For each vertex v in x, the sub-lattice of reachable vertices from v is constructed and converted to a tree z of type t`%T`.
- The function f is applied to the pair (i, z), where i is a list of inheritances computed from previous evaluations of f. When visiting the root node, i is the empty list.
- The function f returns a pair (w, b) where w becomes the corresponding vertex to v in the output lattice y, and b is a list of bequests.
  - The number of bequests in b (i.e., its length) must be equal to the number of descendents of z (i.e., the length of `~&v` z) or else an exception is raised with a diagnostic message of "`bad forward inducer`".
  - The bequests from each ancestor of each descendent of z are collected automatically into the inheritances to be passed to f when the descendent is visited.

The example of forward induction in Listing 18.1 demonstrates the general form of an algorithm to compute all possible paths from the root to each vertex in a lattice. This type of problem might occur in practice for valuing path dependent financial derivatives. The argument to the `fwi` combinator

```
^rlrDlShiX2lNXQ\~&rv ~&l?\~&rdNCNC ~&rdPlLPDrlNCTS
```

takes an argument (i, z) in which z is tree of characters derived from the input lattice, and i is a list of lists of paths, each being inherited from a different ancestor. If i is empty, the list of the singleton list of the root of z is constructed by `~&rdNCNC`, but otherwise, i is flattened to a list of paths and the root of z is appended to each path by

`~&rdPlLPDrlNCTS`. The pair returned by this function (w, b) has a copy of this result as w, and a list of copies of it in b, with one for each descendent of z.

The `paths` function using this forward induction algorithm in Listing 18.1 can be demonstrated as follows.

```
$ fun lat lax --m="paths x" --c %sLG
<
   [0:0: <'a'>^: <1:0,1:1>],
   [
      1:1: <'ac'>^: <2:0,2:1,2:2>,
      1:0: <'ab'>^: <2:0,2:1,2:2>],
   [
      2:2: <'abf','acf'>^: <2:0,2:1,2:2,2:3>,
      2:1: <'abe','ace'>^: <2:0,2:1,2:2,2:3>,
      2:0: <'abd','acd'>^: <2:0,2:1,2:2,2:3>],
   [
      2:3: <'abdj','acdj','abej','acej','abfj','acfj'>^: <>,
      2:2: <'abdi','acdi','abei','acei','abfi','acfi'>^: <>,
      2:1: <'abdh','acdh','abeh','aceh','abfh','acfh'>^: <>,
      2:0: <'abdg','acdg','abeg','aceg','abfg','acfg'>^: <>]>
```

As this example suggests, some pruning may be required in practice to limit the inevitable combinatorial explosion inherent in computing all possible paths within a larger lattice.

swi

A function of the form `swi` f takes a lattice x of type t`%G` as input, and returns an isomorphic lattice y of type u`%G`. Each vertex w in y is given by $f(s, v)$ where v is the corresponding vertex in x, and s is the ordered list of vertices on the level of v.

The `swi` combinator is mnemonic for "sideways induction". An example with the function `^H\~&r -$+ ~&lizyCX` shown in Listing 18.1 rolls each level of the lattice by constructing a finite map (`-$`) from each vertex to its successor in the list of siblings.

```
$ fun lat lax --m="roll x" --c %cG
<
   [0:0: `a^: <1:0,1:1>],
   [
      1:1: `b^: <2:0,2:1,2:2>,
      1:0: `c^: <2:0,2:1,2:2>],
   [
      2:2: `e^: <2:0,2:1,2:2,2:3>,
      2:1: `d^: <2:0,2:1,2:2,2:3>,
      2:0: `f^: <2:0,2:1,2:2,2:3>],
   [
      2:3: `i^: <>,
      2:2: `h^: <>,
```

```
   2:1: `g^: <>,
   2:0: `j^: <>]>
```

fswi

This combinator provides the most general form of induction pattern on lattices, allowing functional dependence of each vertex on ancestors and siblings. Given a lattice x of type t`%G`, the function `fswi` f returns an isomorphic lattice y of type u`%G`.

- For each vertex v in x, the sub-lattice of reachable vertices from v is constructed and converted to a tree z of type t`%T`.
- The function f is applied to the tuple $((i, s), z)$, where i is a list of inheritances computed from previous evaluations of f, and s is the ordered list of vertices in x on the level of v. When visiting the root node, i is the empty list.
- The function f returns a pair (w, b) where w becomes the corresponding vertex to v in the output lattice y, and b is a list of bequests.
  - The number of bequests in b (i.e., its length) must be equal to the number of descendents of z (i.e., the length of `~&v` z) or else an exception is raised with a diagnostic message of "`bad forward inducer`".
  - The bequests from each ancestor of each descendent of z are collected automatically into the inheritances to be passed to f when the descendent is visited.

The example in Listing 18.1 shows how a lattice can be constructed in which each vertex stores a list of lists of neighboring vertices $\langle a, u, l, d\rangle$ with the ancestors, upper sibling, lower sibling, and descendents of the corresponding vertex in the input lattice.

```
$ fun lat lax --m="neighbors x" --c %sLG
<
   [0:0: <'','','','bc'>^: <1:0,1:1>],
   [
      1:1: <'a','','b','def'>^: <2:0,2:1,2:2>,
      1:0: <'a','c','','def'>^: <2:0,2:1,2:2>],
   [
      2:2: <'bc','','e','ghij'>^: <2:0,2:1,2:2,2:3>,
      2:1: <'bc','f','d','ghij'>^: <2:0,2:1,2:2,2:3>,
      2:0: <'bc','e','','ghij'>^: <2:0,2:1,2:2,2:3>],
   [
      2:3: <'def','','i',''>^: <>,
      2:2: <'def','j','h',''>^: <>,
      2:1: <'def','i','g',''>^: <>,
      2:0: <'def','h','',''>^: <>]>
```

But then if we do not ever take time, how can we ever have time?

The Merovingian in *The Matrix Reloaded*

19 Time keeping

A small library of functions, `stt`, exists for the purpose of converting calendar times between character strings and natural number representations.

`one_time`

the constant character string `'Fri Mar 18 01:58:31 UTC 2005'`

`string_to_time`

This function takes a character string representing a time and returns the corresponding number of seconds since midnight, January 1, 1970, ignoring leap seconds.

- The input format is "`Thu, 31 May 2007 19:01:34 +0100`".
- The year must be 1970 or later.
- If the time zone offset is omitted, universal time is assumed.
- The fields can be in any order provided they are separated by one or more spaces.
- Commas are treated as spaces.
- The day of the week is ignored and can be omitted.
- Time zone abbreviations such as `GMT` are allowed but ignored.
- Month names must be three letters, and can be all upper or all lower case, in addition to the mixed case format shown.

`time_to_string`

> This function takes a natural number of non-leap seconds since midnight, January 1, 1970 and returns a character string expressing the corresponding date and time. The output format is "`Thu May 31 17:50:01 UTC 2007`".

The following example shows the moments when POSIX time was a power of two.

```
$ fun stt --m="time_to_string* next31(double) 1" --s
Thu Jan  1 00:00:01 UTC 1970
Thu Jan  1 00:00:02 UTC 1970
Thu Jan  1 00:00:04 UTC 1970
Thu Jan  1 00:00:08 UTC 1970
Thu Jan  1 00:00:16 UTC 1970
Thu Jan  1 00:00:32 UTC 1970
Thu Jan  1 00:01:04 UTC 1970
Thu Jan  1 00:02:08 UTC 1970
Thu Jan  1 00:04:16 UTC 1970
Thu Jan  1 00:08:32 UTC 1970
Thu Jan  1 00:17:04 UTC 1970
Thu Jan  1 00:34:08 UTC 1970
Thu Jan  1 01:08:16 UTC 1970
Thu Jan  1 02:16:32 UTC 1970
Thu Jan  1 04:33:04 UTC 1970
Thu Jan  1 09:06:08 UTC 1970
Thu Jan  1 18:12:16 UTC 1970
Fri Jan  2 12:24:32 UTC 1970
Sun Jan  4 00:49:04 UTC 1970
Wed Jan  7 01:38:08 UTC 1970
Tue Jan 13 03:16:16 UTC 1970
Sun Jan 25 06:32:32 UTC 1970
Wed Feb 18 13:05:04 UTC 1970
Wed Apr  8 02:10:08 UTC 1970
Tue Jul 14 04:20:16 UTC 1970
Sun Jan 24 08:40:32 UTC 1971
Wed Feb 16 17:21:04 UTC 1972
Wed Apr  3 10:42:08 UTC 1974
Tue Jul  4 21:24:16 UTC 1978
Mon Jan  5 18:48:32 UTC 1987
Sat Jan 10 13:37:04 UTC 2004
```

I wish you could see what I see.

Neo in *The Matrix Revolutions*

20

Data visualization

A library named `plo` for plotting graphs of real valued functions along the lines of Figures 15.1 and 15.2 is documented in this chapter. Features include linear, logarithmic and non-numeric scales, variable line colors and styles, arbitrary rotation of axis labels, inclusion of LaTeX code fragments as annotations, scatter plots, and piecewise linear plots. More sophisticated curve fitting can be achieved by using this library in combination with the `fit` library documented in Chapter 14.

The main advantages of this library are that it allows data visualization to be readily integrated with with numerical applications developed in Ursala, and the results generated in LaTeX code will match the fonts of the document or presentation in which they are included. The intention is to achieve publication quality typesetting.

20.1 Functions

A plot is normally specified in its entirety by a record data structure which is then translated as a unit to LaTeX code by the following functions.

plot

Given a record of type `_visualization`, this function returns a LaTeX code fragment as a list of character strings that will generate the specified plot.

In order for a plot generated by this function to be typeset in a LaTeX document, the document preamble must contain at least these lines.

```
\usepackage{pstricks}
\usepackage{pspicture}
\usepackage{rotating}
```

Listing 20.1 a nearly minimal example of a plot

```
#import std
#import plo

#output dot'tex' plot

f =

visualization[
   curves: <curve[points: <(0.,0.),(1.,1.),(2.,-1.),(3.,0.)>]>]
```

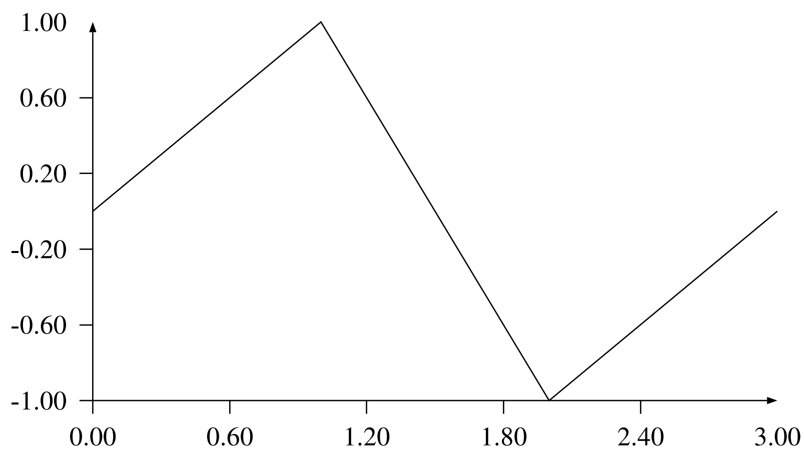


Figure 20.1: an unlabeled plot with default settings generated from Listing 20.1

It is also recommended to include the command

```
\psset{linewidth=.5pt,arrowinset=0,arrowscale=1.1}
```

near the beginning of the document after the `\begin{document}` command.

An example demonstrating the `plot` function is shown in Listing 20.1, and the resulting plot in Figure 20.1. In practice, the points in the plot are more likely to be algorithmically generated than enumerated as shown, but it is often appropriate to use the `plot` function as a formatting function in an `#output` directive. Doing so allows the LaTeX file to be generated as follows.

```
$ fun plo plex.fun
fun: writing `f.tex'
```

where `plex.fun` is the name of the file containing Listing 20.1. The plot stored in `f.tex` can then be used in another document by the LaTeX command `\input{f}`. The `visualization` record structure used in this example is explained in the next section.

latex_document

This function wraps a given a LaTeX code fragment in some additional code to allow it to be processed as a free standing document.

An attempt to typeset the output from the `plot` function by the shell command such as

```
$ latex f.tex
```

will be unsuccessful because a LaTeX document requires some additional front matter that is not part of the output from the `plot` function. The `latex_document` function solves this problem by incorporating the commands mentioned above in the output, among others. A typical usages would be

```
f = latex_document plot visualization[...]
```

or similar variations involving the `#output` directive. The result can be typeset on its own but not included into another document. This function is useful mainly for testing, because in practice the code for a plot is more likely to be included into another document.

20.2 Data structures

A basic vocabulary of useful concepts for describing a plot is as follows.

- A planar cartesian coordinate system denominated in points, where 1 inch $=$ 72 points, fixes any location with respect to the plot
- The rectangular region of the plane bounded by the extrema of the axes in the plot is known as the viewport.
  - The dimensions of the viewport are (v_x, v_y).
  - The lower left corner is at coordinates $(0, 0)$.
- A somewhat larger rectangular region sufficient to enclose the viewport and the labels of the axes is known as the bounding box.
  - Dimensions of the bounding box are (b_x, b_y).
  - The lower left corner is at coordinates (c_x, c_y).
- Some additional dimensions in the plot are
  - the space at the top, $h = b_y + c_y - v_y$
  - the space on the right, $m = b_x + c_x - v_x$
- Numerical values relevant to the functions being plotted are scaled and translated to this coordinate system.

visualization

This function is the mnemonic for a record used to specify a plot for the `plot` function. The fields in the record have these interpretations in terms of the above notation. All numbers are in units of points.

- `viewport` – the pair of floating point numbers (v_x, v_y)
- `picture_frame` – the pair of pairs $((b_x, b_y), (c_x, c_y))$
- `headroom` – space above the viewport, $h = b_y + c_y - v_y$
- `margin` – space to the right of the viewport, $m = b_x + c_x - v_x$
- `abscissa` – a record of type `_axis` that describes the horizontal axis
- `pegaxis` – a record of type `_axis` describing a second independent axis
- `ordinates` – a list of one or two records describing the vertical axes
- `curves` – a list of records of type `_curve` specifying the data to be plotted
- `boxed` – a boolean value causing the bounding box to be displayed when true

In a planar plot, there is no need for a second independent axis, so the `pegaxis` field is ignored by the `plot` function. The data structures for axes and curves are explained shortly, but some further notes on the numeric dimensions in the `visualization` record are appropriate.

- If no value is specified for the `headroom`, a default of 25 points is used.
- If no value is specified for the `margin`, a default value of 10 points is used if there is one vertical axis, and 30 points is used of there are two.
- Default values of b_x and b_y are 300 and 200 points.
- Default values of c_x and c_y are both -32.5 points.
- The `viewport` is always determined automatically by the other dimensions.

The default values of h and m are usually adequate, but they are only approximate. Their optimum values depend on the width or height of the text used to label the axes. If the margins are too small or too large, the plot may be improperly positioned on the page. In such cases, the only remedy is to use the `boxed` field to display the bounding box explicitly, and to adjust the margins manually by trial and error until the outer extremes of the labels coincide with its boundaries. After the right dimensions are determined, the bounding box can be hidden for the final version.

The functions depicted in a plot can be real valued functions of real variables, or they can depend on discrete variables of unspecified types represented as series of character strings. The data structure for an axis accommodates either alternative.

`axis`

This function is the mnemonic for a record describing an axis, which is used in several fields of the `visualization` record. This type of record has the following fields.

- `variable` – a character string containing a LaTeX code fragment for the main label of the axis, usually the name of a variable
- `alias` – a pair of floating point numbers (dx, dy) describing the displacement in points of the `variable` from its default position
- `hats` – a list of character strings or floating point numbers to be displayed periodically along the axis
- `rotation` – the counter-clockwise angular displacement measured in degrees whereby the `hats` are rotated from a horizontal orientation
- `hatches` – a list of character strings or floating point numbers determining the coordinate transformation
- `intercept` – a list containing a single floating point number or character string identifying a point where the axis crosses an orthogonal axis
- `placer` – function that maps any value along the continuum or discrete space associated with the axis to a floating point number in the range $0 \dots 1$.

The coordinate transformation implied by the `placer` normally doesn't have to be indicated explicitly, because it is inferred automatically from the `hatches` field.

- If the `hatches` field consists of a sequence of non-numeric values $\langle s_0 \dots s_n \rangle$, then the `placer` function is that which maps s_i to i/n.
- If the `hatches` are a sequence of floating point numbers $\langle x_0 \dots x_n \rangle$ for which $x_{i+1} - x_i$ is constant within a small tolerance, then the `placer` function maps any given x to $(x - x_0)/(x_n - x_0)$.
- If the `hatches` are a sequence of positive floating point numbers $\langle x_0 \dots x_n \rangle$ for which x_{i+1}/x_i is constant within a small tolerance, the `placer` function maps any given x to $(\ln x - \ln x_0)/(\ln x_n - \ln x_0)$.
- For other sequences of floating point numbers, the `placer` function performs linear interpolation.

However, if a value for the `placer` field is specified by the user, it is employed in the coordinate transformation. The `axis` record has several other automatic initialization features.

- Zero values are inferred for unspecified `rotation` and `alias`.

- If the intercept is unspecified, the plot function positions an axis on the viewport boundary.
- If the hats field is unspecified, it is determined from the hatches field.
  - Symbolic hatches (i.e., character strings) are copied verbatim to the hats field.
  - Numeric hatches are translated to character strings either in fixed or scientific notation, depending on the dynamic range.
- If the hatches field is not specified but the hats field is a list of strings in fixed or exponential notation, the hatches field is read from it using the math..strtod library function.

When the axis forms part of a visualization record, further initialization of the hatches field is performed automatically, because its values are implied by the curves.

curve

This function is the mnemonic for a record data structure representing a curve to be plotted, of which there are a list in the curves field of a visualization record. The curve record has the following fields.

- points – a list of pairs $\langle (x_0, y_0) \dots (x_n, y_n) \rangle$ representing the data to be plotted, where x_i and y_i can be character strings or floating point numbers
- peg – a value that's constant along the curve if it's a function of two variables
- attributes – a list of assignments of attributes to keywords recognized by the LaTeX pstricks package to describe line colors and styles
- decorations – a list of triples $\langle ((x_0, y_0), s_0) \dots ((x_n, y_n), s_n) \rangle$ where x_i and y_i are coordinates consistent with the points field indicating the placement of a LaTeX code fragment s_i on the plot, where s_i is a list of character strings
- scattered – a boolean value causing the points not to be connected when plotted if true
- discrete – a boolean value causing points to be disconnected and also causing each point to be plotted atop a vertical line if true
- ordinate – a pointer (e.g., &h or &th) with respect to the ordinates field in a visualization record that identifies the vertical axis whose placer is used to transform the y values in the points field

Some additional notes on these fields:

- The default value for the ordinate field is &h, which is appropriate when there is a single vertical axis.

Listing 20.2 demonstration of decorations, attributes, and axes

```
#import std
#import plo
#import flo

#output dot'tex' plot

plop =

visualization[
   picture_frame: ((400.,300.),()),
   abscissa: axis[
      hats: printf/*'%0.2f' ari13/0. 3.,
      variable: 'time ($\mu s$)'],
   ordinates: <
      axis[variable: 'feelgood factor (erg$/$lightyear$^2$)']>,
   curves: <
      curve[points: <(0.,0.),(1.,1.),(2.,-1.),(3.,0.)>],
      curve[
         decorations: ~&iNC/(0.35,-0.6) -[
            \begin{picture}(0,0)
            \psset{linecolor=black}
            \psline{-}(0,0)(10,0)
            \put(15,0){\makebox(0,0)[l]{\textsl{realized}}}
            \psset{linecolor=lightgray}
            \psline{-}(0,20)(10,20)
            \put(15,20){\makebox(0,0)[l]{\textsl{projected}}}
            \put(-10,-15){\dashbox(75,50){}}
            \end{picture}]-,
         attributes: <'linecolor': 'lightgray'>,
         points: <(0.,0.),(3.,1.5)>]>]
```

- In a planar plot, the `peg` field is ignored.
- If the `attributes` field contains assignments `<'foo': 'bar'...>`, they are passed through as `\psset{foo=bar...}`.
- The assigned `attributes` apply cumulatively to subsequent curves in the list of `curves` in a `visualization` record.

The `psset` command is documented in the `pstricks` reference manual. Frequently used attributes are `linecolor` and `linewidth`.

20.3 Examples

A possible way of using this library without reading all of the preceding documentation is to copy one of the examples from this section and modify it to suit, referring to the

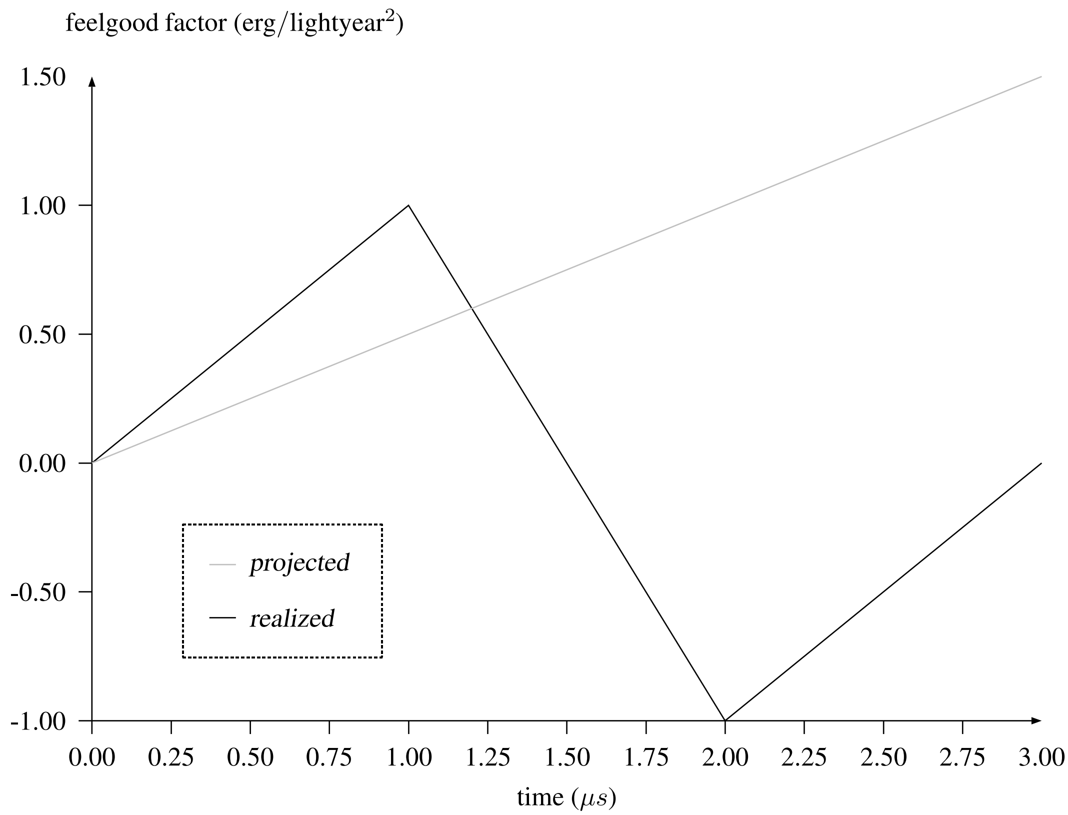


Figure 20.2: output from Listing 20.2

Listing 20.3 symbolic axes, rotation, margins, discrete curves, generated data, and interpolation

```
#import std
#import nat
#import plo
#import flo
#import fit

data = ~&p(ari7/0. 1.,rand* iota 7)

#output dot'tex' plot

slam =

visualization[
   margin: 35.,
   picture_frame: ((400.,300.),((),-75.)),
   abscissa: axis[
      rotation: -60.,
      hats: <
         'impulse',
         'light speed',
         'ludicrous speed',
         'ridiculous speed'>,
      variable: 'velocity ($v$)'],
   ordinates: ~&iNC axis[
      hatches: ari11/0. 1.,
      variable: 'tunneling probability ($\rho$)'],
   curves: <
      curve[discrete: true,points: data],
      curve[
         points: ^(~&,sinusoid data)* ari200/0. 1.,
         attributes: <'linecolor': 'lightgray'>]>]
```

documentation only as needed. Most of the features are exemplified at one point or another.

Listing 20.2 demonstrates multiple curves with different attributes, and user-written LaTeX code decorations inserted "inline". Note that the coordinates of the decorations are in terms of those of the curve, rather than being absolute point locations, so they will scale automatically if the bounding box size is changed. The results are shown in Figure 20.2.

Listing 20.3 and the results shown in Figure 20.3 demonstrate an axis with symbolic rather than numeric hatches. In this case, the data are numeric and the axis labels are chosen arbitrarily, but data that are themselves symbolic can also be used. Further features of this example:

- the discrete plotting style, wherein the points are separated from one another but connected to the horizontal axis by vertical lines.
- a smooth curve generated using the `sinusoid` interpolation function from the `fit`

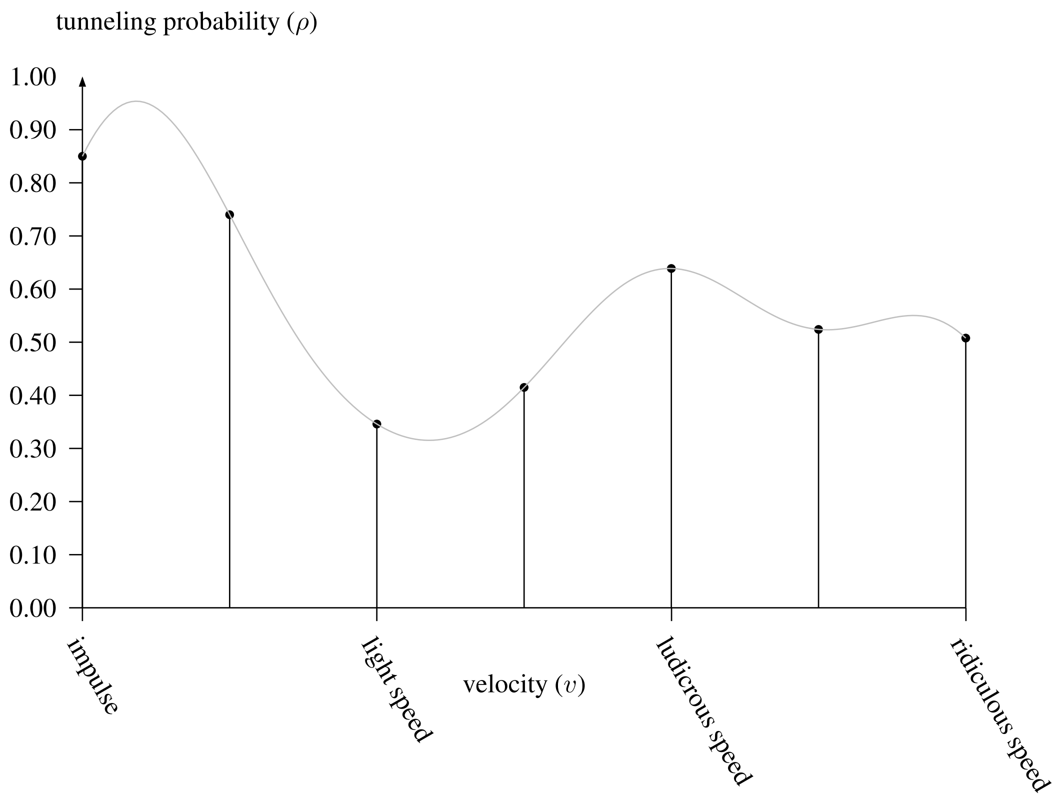


Figure 20.3: output from Listing 20.3

Listing 20.4 aliases, intercepts, margins, and selective hats

```
#import std
#import nat
#import plo
#import flo

#output dot'tex' plot

para =

visualization[
   margin: 25.,
   picture_frame: ((400.,200.),(-10.,-20.)),
   abscissa: axis[
      hats: printf/*'%0.2f' ari9/-1. 1.,
      alias: (205.,27.),
      variable: '$x$'],
   ordinates: ~&iNC axis[
      alias: (8.,0.),
      intercept: <0.>,
      hats: ~&NtC printf/*'%0.2f' ari5/0. 1.,
      variable: '$y$'],
   curves: <curve[points: ^(~&,sqr)* ari200/-1. 1.]>]
```

library documented in Chapter 14

- A rotation of the horizontal axis labels

The scattered plot style is similar to the discrete style but omits the vertical lines.

Listing 20.4 and the results in Figure 20.4 demonstrate some possibilities for positioning axes and labels. The vertical axis is displayed in the center by way of the `intercept`, and the label x of the horizontal axis is displayed to the right rather than below. The zero on the vertical axis is suppressed in the `hats` field of the `ordinate` so as not to clash with the horizontal axis. Some manual adjustment to the margins and bounding box are made based on visual inspection of the bounding box in draft versions.

The last example in Listing 20.5 and Figure 20.5 shows how multiple functions can be plotted on different vertical scales with the same horizontal axis. With two ordinates and two curves, each refers to its own. A logarithmic scale is automatically inferred for the right ordinate because the hatches are given as a geometric progression. A decoration for each curve reduces ambiguity by identifying the function it represents and hence the corresponding vertical axis.

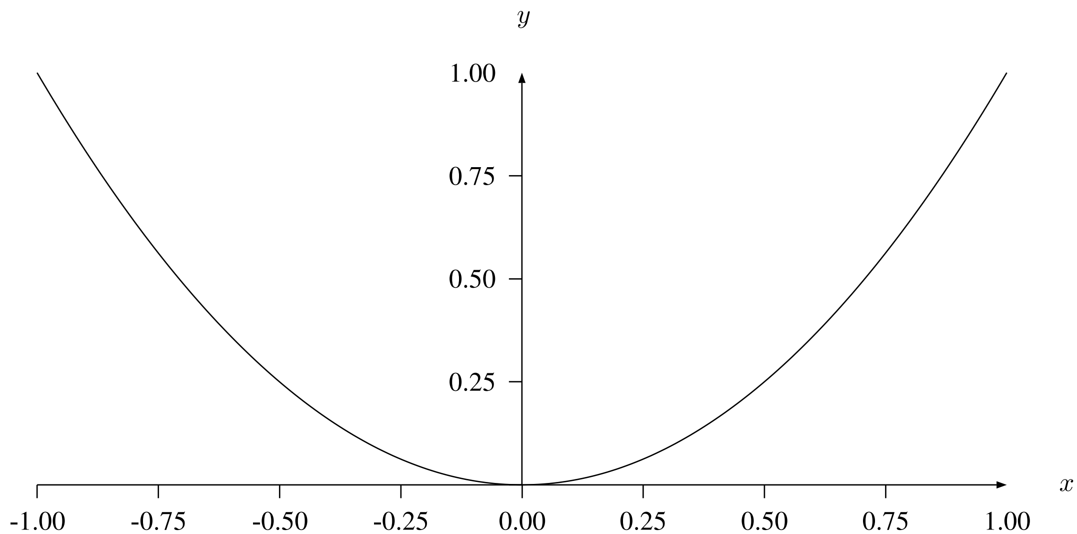


Figure 20.4: textbook style parabola illustration from Listing 20.4

Listing 20.5 logarithmic scales, decorations, and multiple ordinates

```
#import std
#import nat
#import plo
#import flo

#output dot'tex' plot

gam =

visualization[
   picture_frame: ((400.,250.),(-25.,())),
   margin: 50.,
   abscissa: axis[variable: '$x$',hats: ~&hS %nP* ~&tt iota 7],
   ordinates: <
      axis[variable: '$\Gamma''(x)$',hats: printf/*'%0.1f' ari6/0. 2.],
      axis[variable: '$\Gamma(x)$',hatches: geo6/1. 120.]>,
   curves: <
      curve[
         ordinate: &h,
         decorations: <((2.8,1.0),-[$\Gamma'$]-)>,
         points: ^(~&,rmath..digamma)* ari200/2. 6.],
      curve[
         ordinate: &th,
         decorations: <((4.8,10.),-[$\Gamma$]-)>,
         points: ^(~&,rmath..gammafn)* ari200/2. 6.]>]
```

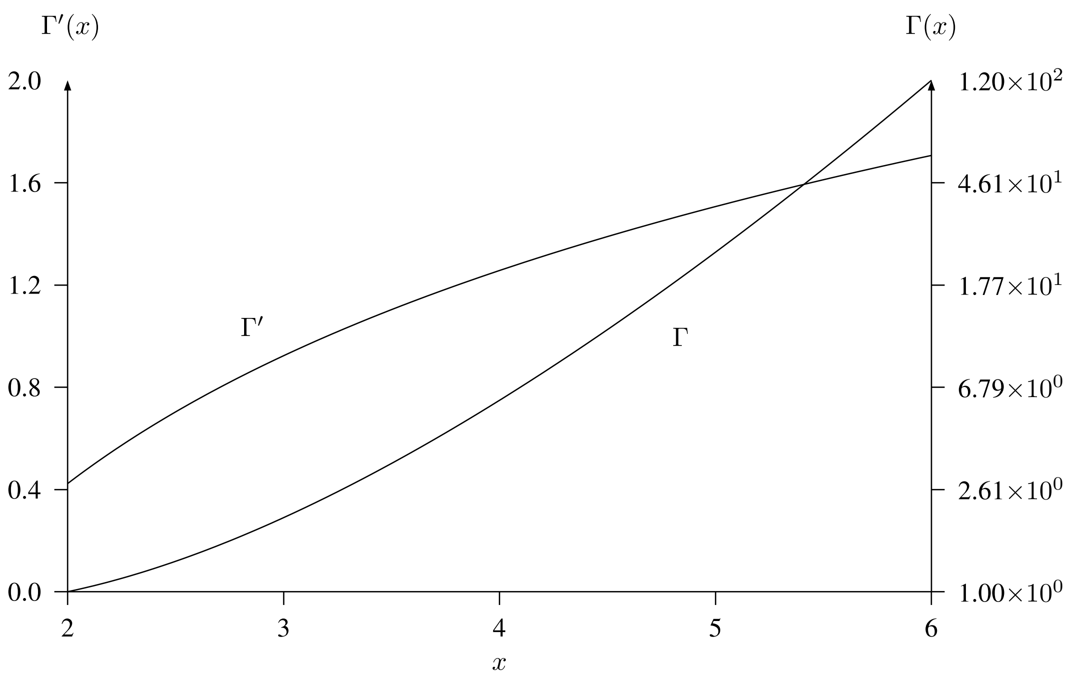


Figure 20.5: gamma and digamma function plots with different vertical scales from Listing 20.5

It's a way of looking at that wave and saying "Hey Bud, let's party".

Sean Penn in *Fast Times at Ridgemont High*

21 Surface rendering

Following on from the previous chapter, a library called `ren` uses the same data structures to depict functions of two variables graphically as surfaces. The rendering algorithm features correct perspective and physically realistic shading of surface elements based on a choice of simulated semi-diffuse light sources. The renderings are generated as LaTeX code depending on the `pstricks` package, so that hidden surface removal is accomplished by the back end Postscript rendering engine. The user has complete control over the choice of a focal point, and scaling of the image both in the image plane and in 3-space.

21.1 Concepts

To depict a function of two variables as a surface, a specification needs to be given not only of the function, but of certain other characteristics of the image. These include its focal point relative to a hypothetical three dimensional space, which can be understood as the position of an observer or a simulated camera viewing the surface, and the position of a simulated light source. Regardless of its relevance to the data, shading consistent with a light source is necessary for visual perception. There are also the same requirements for specifying the axis labels and hatches as in a two dimensional plot. The conventions whereby this information is specified are documented in this section.

21.1.1 Eccentricity

A function $f : \mathtt{R}^2 \rightarrow \mathtt{R}$ defined on a region $[a_0, a_1] \times [b_0, b_1]$ is depicted as a surface confined to the cube with corners $\{0, 1\}^3$ in a right handed cartesian coordinate system. Each input (x, y) in the region is associated with a point in the unit square on the horizontal plane, and the value of $f(x, y)$ is indicated by the height of the surface above that point.

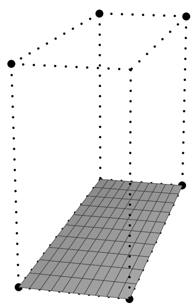
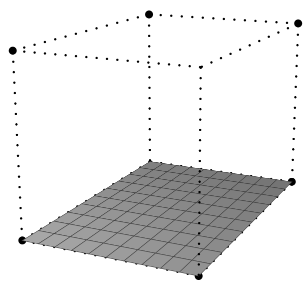
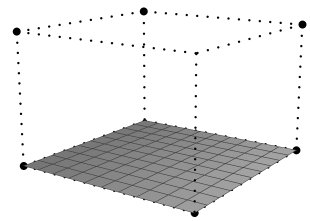
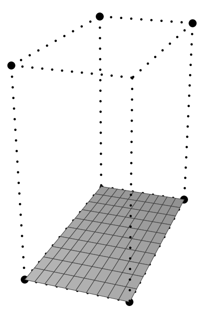
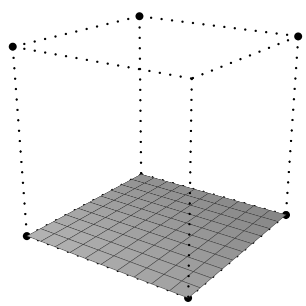
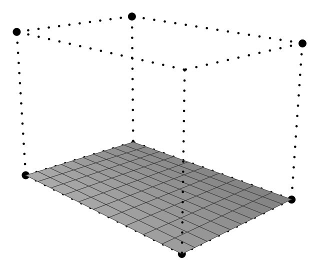
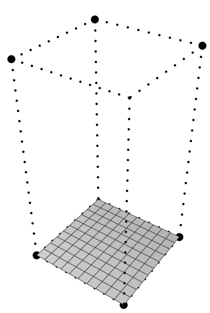
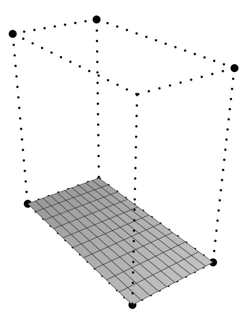
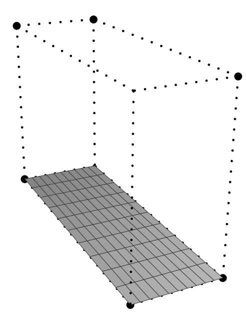
	$x < 1$	$x = 1$	$x > 1$
$y > 1$			
$y = 1$			
$y < 1$			

Table 21.1: eccentricity settings as seen from `ols+`, with origin left and x axis in the foreground

	coordinates			angle (deg.)			coordinates		
code	x	y	z	θ	ϕ	code	x	y	z
`ile+`	2.040	1.578	1.184	35	20	`ole+`	3.194	2.386	1.697
`ime+`	1.842	1.440	1.647	35	35	`ome+`	2.849	2.144	2.508
`ihe+`	1.553	1.237	2.032	35	50	`ohe+`	2.343	1.790	3.181
`iln-`	1.578	2.040	1.184	55	20	`oln-`	2.386	3.194	1.697
`imn-`	1.440	1.842	1.647	55	35	`omn-`	2.144	2.849	2.508
`ihn-`	1.237	1.553	2.032	55	50	`ohn-`	1.790	2.343	3.181
`iln+`	-0.578	2.040	1.184	125	20	`oln+`	-1.386	3.194	1.697
`imn+`	-0.440	1.842	1.647	125	35	`omn+`	-1.144	2.849	2.508
`ihn+`	-0.237	1.553	2.032	125	50	`ohn+`	-0.790	2.343	3.181
`ilw-`	-1.040	1.578	1.184	145	20	`olw-`	-2.194	2.386	1.697
`imw-`	-0.842	1.440	1.647	145	35	`omw-`	-1.849	2.144	2.508
`ihw-`	-0.553	1.237	2.032	145	50	`ohw-`	-1.343	1.790	3.181
`ilw+`	-1.040	-0.578	1.184	-145	20	`olw+`	-2.194	-1.386	1.697
`imw+`	-0.842	-0.440	1.647	-145	35	`omw+`	-1.849	-1.144	2.508
`ihw+`	-0.553	-0.237	2.032	-145	50	`ohw+`	-1.343	-0.790	3.181
`ils-`	-0.578	-1.040	1.184	-125	20	`ols-`	-1.386	-2.194	1.697
`ims-`	-0.440	-0.842	1.647	-125	35	`oms-`	-1.144	-1.849	2.508
`ihs-`	-0.237	-0.553	2.032	-125	50	`ohs-`	-0.790	-1.343	3.181
`ils+`	1.578	-1.040	1.184	-55	20	`ols+`	2.386	-2.194	1.697
`ims+`	1.440	-0.842	1.647	-55	35	`oms+`	2.144	-1.849	2.508
`ihs+`	1.237	-0.553	2.032	-55	50	`ohs+`	1.790	-1.343	3.181
`ile-`	2.040	-0.578	1.184	-35	20	`ole-`	3.194	-1.386	1.697
`ime-`	1.842	-0.440	1.647	-35	35	`ome-`	2.849	-1.144	2.508
`ihe-`	1.553	-0.237	2.032	-35	50	`ohe-`	2.343	-0.790	3.181

Table 21.2: observer coordinates and angular displacements from the center of the unit cube

Whereas a cube is normally envisioned as in the center of Table 21.1, the user is also at liberty to emphasize particular dimensions by elongating it in one direction or another. A so called eccentricity given by a pair of floating point numbers (x, y) has $x = y = 1$ for a neutral appearance, both dimensions greater than one for an apparent pizza box shape, both less than one for a tower, and different combinations for other rectangular prisms. The cube is transformed to a box with edges in the ratios of $x : y : 1$ bounded by the origin, and the surface is scaled accordingly.

21.1.2 Orientation

The surface is always rendered from the point of view of an observer looking directly at the center of the prism described above, regardless of its eccentricity, but the position of the observer is a tunable parameter with three degrees of freedom. The position can be specified in principle by its cartesian coordinates, but it is convenient to encode frequently used families of coordinates as shown in Table 21.2.

A specification of observer coordinates for one of these standard positions is a string of the form

$$\texttt{[i|o] [l|m|h] [e|n|w|s] [+|-]}$$

- The first field, mnemonic for "in" or "out" determines the zoom, which is the distance of the observer from the center of the cube. The image is scaled to the same size regardless of the distance, but the inner position results in more pronounced apparent convergence of parallel lines due to perspective.
- The second field, mnemonic for "low", "medium" or "high", refers to the angle of elevation. The angle is formed by the vector from the center of the cube to the observer with the horizontal plane. These angles are defined as $20°$, $35°$, and $50°$, respectively.
- The third field, mnemonic for "east", "north", "west" or "south", indicates the approximate lateral angular displacement of the observer, with `e` referring to the positive x direction, and `n` referring to the positive y direction.
- Because it is less visually informative to sight orthogonally to the axes, the last field of − or + indicates a clockwise or counterclockwise displacement, respectively, of $35°$ from the direction indicated by the preceding field.

The cartesian coordinates shown in Table 21.2 apply only to the case of neutral eccentricity. For oblong boxes, the positions are scaled accordingly to maintain these angular displacements.

The effects of zooms, elevations, and lateral angular displacements are demonstrated in Tables 21.3 and 21.4, with Table 21.4 showing various views of the same quadratic surface.

21.1.3 Illumination

The library provides three alternatives for light source positions in a rendering, which are left, right, and back lighting. The most appropriate choice depends on the shape of the surface being rendered and the location of the observer.

- left lighting postulates a light source above and behind the focal point to the left
- right lighting is based on a source above and behind the focal point to the right
- back lighting simulates a light source facing the observer, slightly to the left and low to the horizon

Best results are usually obtained with either left or right lighting, where more visible surface elements face toward the light source than away from it. Back lighting is suitable only for special effects and will generally result in lower contrast.

An example of each style of lighting is shown in Table 21.5. The central maximum does not cast a shadow on the outer wave, because the image is not a true ray tracing simulation. The shade of each surface element is determined by the angle of incidence with the light source, and to lesser extent by the distance from it.

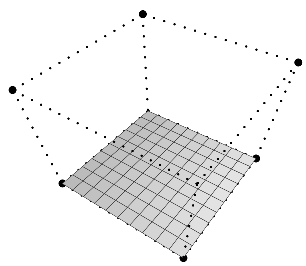
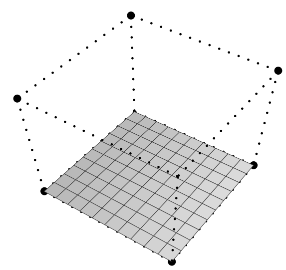
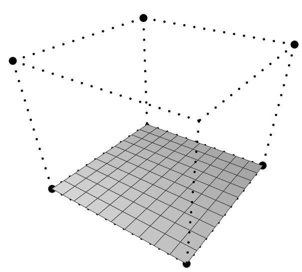
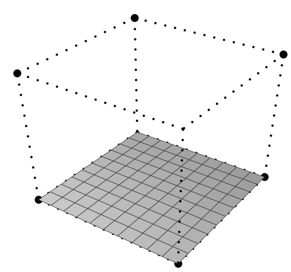
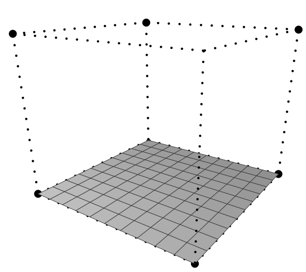
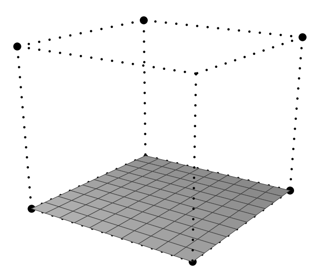
	zoom	
eye level	in	out
high		
medium		
low		

Table 21.3: orthogonal choices of recommended levels and zooms

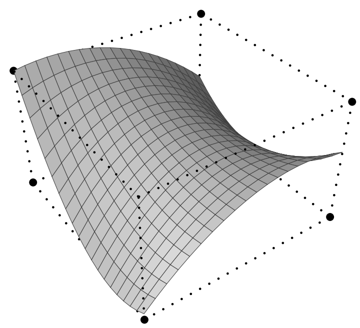
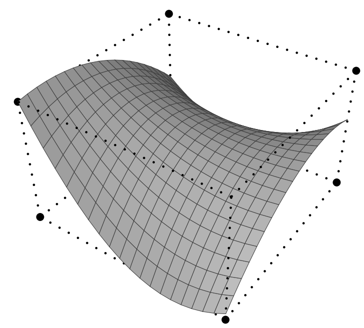
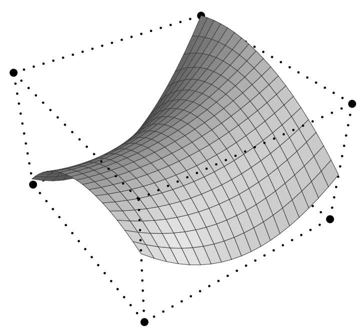
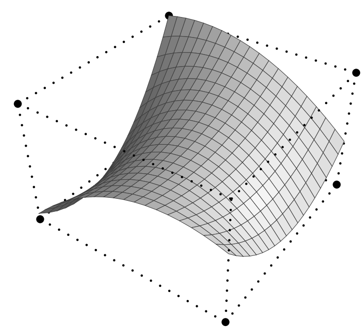
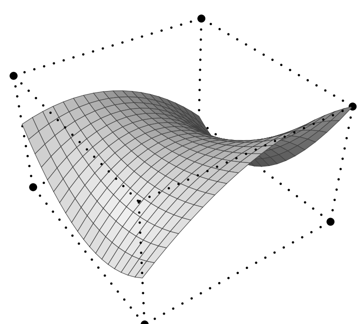
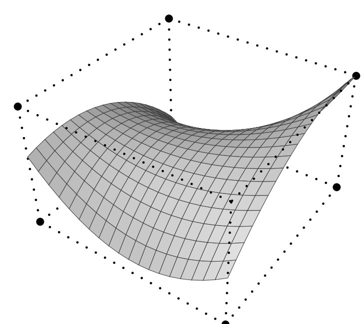
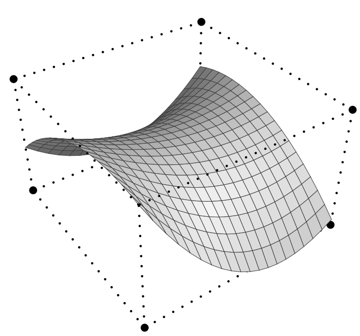
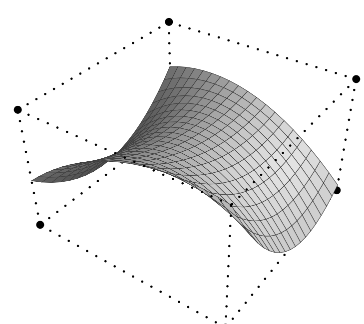
quadrant	+	−
`e+ / n−`		
`n+ / w−`		
`w+ / s−`		
`s+ / e−`		

Table 21.4: visual effects of lateral angular displacements

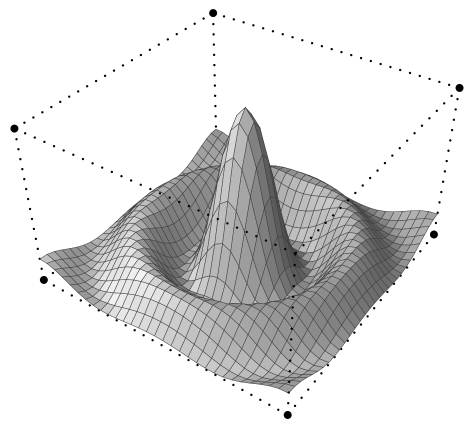
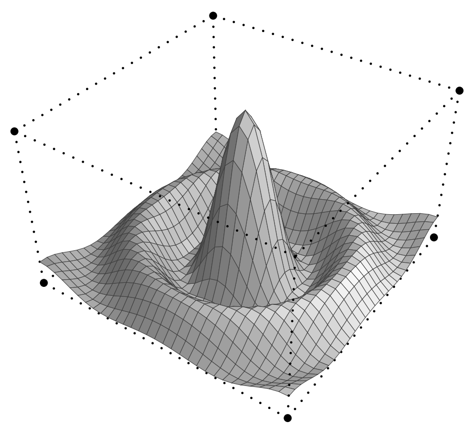
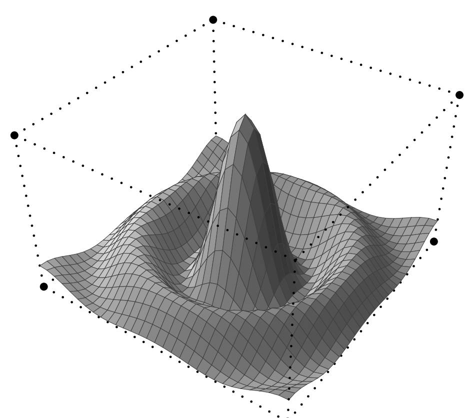
light source	visual effect
left	
right	
back	

Table 21.5: effects of left, right, and back lighting

21.2 Interface

Use of the library is fairly simple when the concepts explained in the previous section are understood.

`left_lit_rendering`

This function takes an argument of the form $((o, e), v)$ to a list of character strings containing the LaTeX code fragment for a surface rendering with the light source to the left.

- o is an observer position specified either as a code from Table 21.2 in a character string, or as absolute cartesian coordinates in a list of three floating point numbers.
- e is either empty or a pair of floating point numbers (x, y) describing the eccentricity of the box in which the surface is inscribed, as explained in Section 21.1.1. If e is empty, neutral eccentricity (i.e., a cube shape) is inferred.
- v is a `visualization` record as documented in the previous chapter specifying axes and the surface to be rendered as a family of curves.
  - The `visualization` record must contain exactly one ordinate axis, an abscissa, and a non-empty peg axis.
  - Each curve in the `visualization` must have the same number of points.
  - The i-th point in each curve must have the same left coordinate across all curves for all i.
  - Each curve must have a `peg` field serving to locate it along the `pegaxis`.

  The abscissa is rendered along the x or “east” axis in 3-space, the peg axis along the y or “north”, and the ordinate along the vertical axis.

`right_lit_rendering`

This function follows the same conventions as the one above but renders the surface with a light source to the right.

`back_lit_rendering`

This function is the same as above but with back lighting.

`rendering`

This function renders the surface with a randomly chosen light source either to the left or to the right.

Most features of the `visualization` record documented in the previous chapter, such as use of symbolic hatches or logarithmic scales, generalize to three dimensional plots as one would expect, other than as noted below.

- The `intercept`, `rotation`, and `attributes` fields are ignored.
- The `discrete` and `scattered` flags are inapplicable.
- The default `picture_frame` is $((400, 400), (-50, -50))$ with the `headroom` and the `margin` at 50 points each.

A square `viewport` field (i.e., with its width equal to its height) is not required but strongly recommended for surface renderings because the image will be distorted otherwise in a way that frustrates visual perception. Any preferred alterations to the aspect ratio should be effected by the eccentricity parameter instead. If the `margin` and `headroom` are equal in magnitude and opposite in sign to the `picture_frame` coordinates and the picture frame is square, as in the default setting above, then the `viewport` will be initialized to a square. Otherwise, the `viewport` should be initialized as such explicitly by the user.

drafts

This function takes a pair (e, v) to a complete LaTeX document represented as a list of character strings containing renderings of a surface from all focal points listed in Table 21.2, with one per page. The parameter e is either an eccentricity (x, y) as explained in Section 21.1.1 or empty, with neutral eccentricity inferred in the latter case. The parameter v is a visualization describing the surface as explained above.

recommended_observers

This is a constant of type `%seLXL` containing the data in Table 21.2. Each item of the list is a pair with a code such as `'ole+'` on the left and the corresponding cartesian coordinates on the right.

The `recommended_observers` list is not ordinarily needed unless one wishes to construct a non-standard observer position by interpolation or perturbation of a recommended one.

A short example using some of these features is shown in Listing 21.1 and Figure 21.1. Although the family of curves is enumerated in this example, it would usually be generated by an expression such as the following in practice,

```
curve$[peg: ~&hl,points: * ^/~&r f]* ~&iiK0lK2x (ari n)/a b
```

where f is a function taking a pair of floating point numbers to a floating point number.

Listing 21.1 short example of a rendering

```
#import std
#import nat
#import plo
#import ren

#output dot'tex' left_lit_rendering/('ilw+',())

surf =

visualization[
   picture_frame: ((280.,280.),(-55.,-25.)),
   margin: 65.,
   headroom: 35.,
   viewport: (210.,210.),
   abscissa: axis[variable: '$x$',hats: <'0','1','2','3'>],
   pegaxis: axis[variable: '$y$',hatches: <1.,5.,9.>],
   ordinates: <axis[variable: '$z$']>,
   curves: <
      curve[peg: 1.,points: <(0.,2.),(1.,3.),(2.,4.),(3.,5.)>],
      curve[peg: 5.,points: <(0.,1.),(1.,2.),(2.,3.),(3.,4.)>],
      curve[peg: 9.,points: <(0.,0.),(1.,1.),(2.,2.),(3.,3.)>]>]
```

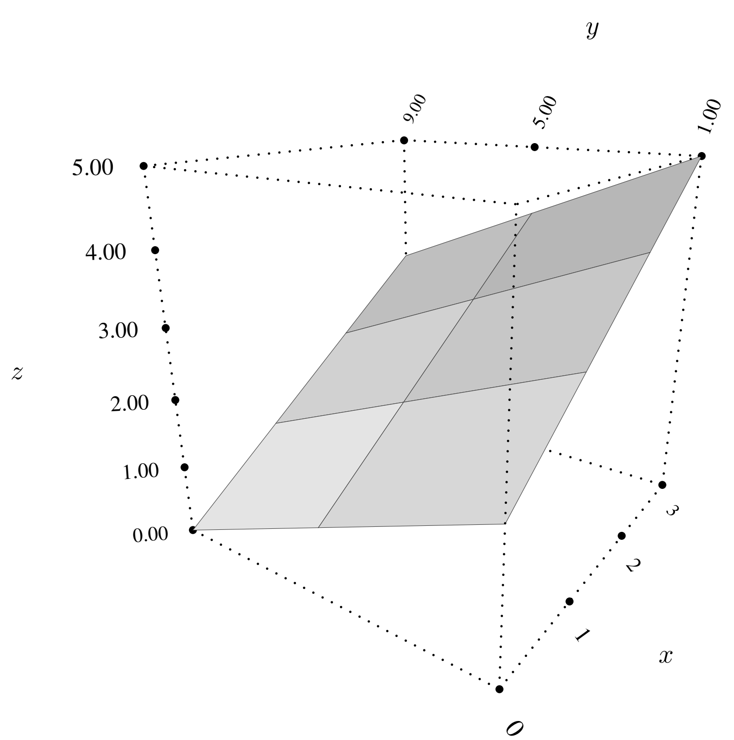


Figure 21.1: output from Listing 21.1

You talkin' to me?

Robert De Niro in *Taxi Driver*

22
Interaction

An unusual and powerful feature of Ursala is its interoperability with command line interpreters such as shells and computer algebra systems. Ready made interfaces are provided for the numerical and statistical packages `Octave`, `R`, and `scilab`, the computer algebra systems `axiom`, `maxima`, and `pari-gp`, and the number theory package `gap`. These interfaces make any interactive function from these packages callable within the language, even if the function is user defined and not included in the package's development library.

There are also interfaces to the standard shells `bash` and `psh` (the `perl` shell), and to privileged shells opened by the `su` command. Orthogonal to the choice of an application package or shell is the option to access it locally or on a remote host via `ssh`.

The above mentioned packages incorporate an extraordinary wealth of mathematical expertise, and with their extensible designs and scripting languages, each is a capable programming platform by itself. However, for a developer choosing to work primarily in Ursala, the value added by the interfaces documented in this chapter is the flexibility to leverage the best features of all of these packages from a single application with a minimum of glue code.

22.1 Theory of operation

The application packages or shells are required to be installed on the local host or the remote host in order to be callable from the language. In the latter case, the remote host needs an `ssh` server and the user needs a shell account in it, but the compiler and virtual machine need only be installed locally. Installation of these applications is a separate issue beyond the scope of this manual, but it is fairly painless at least for Debian and Ubuntu users who are familiar with the `apt-get` utility.

22.1.1 Virtual machine interface

These shells are spawned and controlled at run time by the virtual machine through pipes to their standard input and output streams, as implemented by the `expect` library. Hence, no dynamic loading takes place in the conventional sense. Furthermore, any console output they perform is not actually displayed on the user's console, but recorded by the virtual machine. However, any side effects of executing them persist on the host.

22.1.2 Source level interface

Although a very general class of interaction protocols can be specified in principle, full use demands an understanding of the calling conventions followed by the virtual machine's `interact` combinator as documented in the `avram` reference manual. As an alternative, the functions defined `cli` library documented in this chapter insulate a developer from some of these details for a restricted but useful class of interactions, namely those involving a sequence of commands to be executed unconditionally.

Several options exist for users requiring repetitive or conditional execution of external shell commands. In order of increasing difficulty, they include

- multiple shell invocations with intervening control decisions at the source level
- a user defined command in the application's native scripting language, if any
- a hand coded client/server interaction protocol

22.1.3 Referential transparency

A more complex issue of interaction with external applications is the possible loss of referential transparency.[1] Although the code generated by the `cli` library functions can be invoked and treated in most respects as functions, it is incumbent on the user to recognize and to anticipate the possibility of different outputs being obtained for identical inputs on different occasions. The compiler for its part will detect the `interact` combinator on the virtual code level and refrain from performing any code optimizations depending on the assumption of referential transparency.

22.2 Control of command line interpreters

Several functions concerned with sending commands to a shell and sensing its responses are documented in this section. These are higher order functions parameterized by a data structure of type `_shell` that isolates the application specific aspects of each shell (e.g., syntactic differences between computer algebra systems). The data structure is documented subsequently in this chapter for users wishing to implement interfaces to other applications than those already provided, but may be regarded as an opaque type for the present discussion.

[1] the property of pure functional languages guaranteeing run-time invariance of the semantics of any expression, even those including function calls

22.2.1 Quick start

To invoke and interrogate one of the supported shells on the local host with any sequence of non-interactive commands, the function described below is the only one needed.

ask

This function takes an argument of type `_shell` and returns a function that takes a pair (e, c) containing an environment and a list of commands to a result t containing a list of responses.

- The environment e is list of assignments <n_0: $m_0 \dots$> where each n_i is a character string and each m_i is of a type that depends on the shell.
- The commands c are a list of character strings <$x_0 \dots$> that are recognizable by the shell as valid interactive user input.
- The results t are a list of assignments <x_0: $y_0 \dots$> where each x_i is one of the commands in c, and the corresponding y_i is the result displayed by the shell in response to that command. The y_i value is a list of character strings by default, unless the shell specification stipulates a postprocessor to the contrary.

Most command line interpreters entail some concept of a persistent environment or workspace that can be modeled as a map from identifiers to elements of some application specific semantic domain. The environment is regarded as a passive but mutable entity acted upon by imperative commands. A convention of direct declarative specification of the environment separate from the imperative operations is used by this function in the interest of notational economy. Here are a couple of examples of this function using `bash` as a shell.

```
$ fun cli --m="(ask bash)/<> <'uname','lpq','pwd'>" -c %sLm
<
   'uname': <'Linux'>,
   'lpq': <'hp is ready','no entries'>
   'pwd': <'/home/dennis/fun/doc'>>
$ fun cli --m="(ask bash)/<'a': 'b'> <'echo \$a'>" --c %sLm
<'echo $a': <'b'>>
```

The backslash is needed to quote the dollar sign because this function is being executed from the command line, but normally would not be required.

22.2.2 Remote invocation

The next simplest scenario to the one above is that of a shell or application installed on a remote host. Assuming the host is accessible by `ssh` (the industry standard secure shell protocol), and that the user is an authorized account holder, the following functions allow convenient remote invocation.

hop

> Given a pair of character strings (h, p), where h is a hostname and p is a password, this function returns a function that takes a shell specification of type `_shell` to a result of the same type. The resulting shell specification will call for a remote connection and execution when used as a parameter to the `ask` function.

The host name is passed through to the `ssh` client, so it can be any variation on the form *user*`@`*host*`.`*domain*. An example of how the `hop` function might be used is in the following code fragment.

```
(ask hop('root@kremvax.gov.ru','glasnost') bash)/<> <'du'>
```

Invocations of `hop` can be arbitrarily nested, as in

$$\texttt{hop(}h_0\texttt{,}p_0\texttt{) hop(}h_1\texttt{,}p_1\texttt{) ... hop(}h_n\texttt{,}p_n\texttt{)}\ \langle\textit{shell}\rangle$$

and the effect will be to connect first to h_0, and then from there to h_1, and so on, provided that all intervening hosts have `ssh` clients and servers installed, and the passwords p_i are valid. This technique can be useful if access to h_n is limited by firewall restrictions. However, in such cases it may be more convenient to use the following function.

multihop

> This function, defined as `-++-+ hop*`, takes a list of pairs of host names and passwords `<(`h_0`,`p_0`) ... (`h_n`,`p_n`)>` to a function that transforms an a given shell to a remote shell executable on host h_n through a connection by way of the intervening hosts in the order they are listed.

This function could be used as follows.

$$\texttt{multihop<(}h_0\texttt{,}p_0\texttt{), ... (}h_n\texttt{,}p_n\texttt{)>}\ \langle\textit{shell}\rangle$$

sask

> This function, defined as `ask++ hop`, combines the effect of the `ask` and `hop` functions for a single hop as a matter of convenience. The usage `sask(`h`,`p`)` s is equivalent to `ask hop(`h`,`p`)` s.

22.3 Defined interfaces

As indicated in the previous section, `ask` and related functions are parameterized by a data structure of type `_shell`, which specifies how the client should interact with the application. It also determines the types of objects that may be declared in the application's environment or workspace, and generates the necessary initialization commands and settings. Although a compatible specification for any shell can be defined by the user, some of the most useful ones are defined in the library as a matter of convenience, and documented in this section.

22.3.1 General purpose shells

It is possible for an application in Ursala to execute arbitrary system commands by interacting with a general purpose login shell. When such a shell s is used in an expression of the form $\texttt{(ask } s\texttt{) (<}n_0\texttt{: } m_0 \ldots \texttt{>,} c\texttt{)}$, each m_i value can be either a character string or a list of character strings.

- If m_i is a character string, then an environment variable is implicitly defined by `export` n_i=m_i.
- If m_i is a list of character strings, then a text file is temporarily created in the current working directory with a name of n_i and contents m_i using the standard line editor, `ed`. The text file is deleted when the shell terminates.

There are certain limitations on the commands that may appear in the list c.

- Interactive commands that wait for user input should be avoided because they will cause the client to deadlock.
- Commands using input redirection (for example, "`cat - > file`") also won't work.
- Commands that generate console output generally are acceptable, but they may confuse the client if they output a shell prompt (`$`) at the beginning of a line.

bash

This shell represents the standard GNU command line interpreter of the same name. Some examples using `bash` are given in Section 22.2.1.

psh

This shell is similar to `bash` but provides some additional features to the commands by allowing them to include `perl` code fragments. Please refer to the `psh` home pages at `http://www.focusresearch.com/gregor/psh/index.html` for more information.

su

This function takes a pair of character strings (u, p) representing a user name and password. It returns a shell similar to `bash` but that executes with the account and privileges of the indicated user. If the user name is empty, `root` is assumed.

The following example demonstrates the usage of `su`.

```
$ fun cli -m="(ask su/0 'Z10N0101')/<> <'whoami'>" -c %sLm
<'whoami': <'root'>>
```

If an application is already executing as `root`, it should not attempt to use a shell generated by the `su` function, because such a shell relies on the assumption that it will be prompted for a password. However, any application running as `root` can achieve the same effect just by executing `su` ⟨*username*⟩ as an ordinary shell command.

22.3.2 Numerical applications

The numerical applications whose interfaces are described in this section include linear algebra functions involving vectors and matrices of numbers. Facilities are provided for automatic initialization of these types of variables in the application's workspace.

- When a shell s interfacing to a numerical application is used in an expression of the form `(ask` s`)(<`n_0`:` m_0`...>,`c`)`, each m_i value can be a number, a list of numbers, or a lists of lists of numbers, and will cause a variable to be initialized in the application's workspace that is respectively a scalar, a vector, or a matrix.
- Different numeric types are supported depending on the application, including natural, rational, floating point, and arbitrary precision numbers in the `mpfr` (`%E`) representation. The type is detected automatically.
- If the application supports them, vectors and matrices of character strings are similarly recognized, and may be initialized either as quoted strings or symbolic names depending on the application.
- If an application supports vectors of strings, an attempt is made to distinguish between lists of character strings representing vectors and those representing functions defined in the application's scripting language based on syntactic patterns as documented below. In the latter case, the list of strings is interpreted as the definition of a function and initialized accordingly.

R

This shell pertains to the `R` system for statistical computation and graphics, for which more information can be found at `http://www.R-project.org`. Four types of data can be recognized and initialized as variables in the `R` workspace when this shell is used as a parameter to the `ask` function. Data of type `%e`, `%eL`, and `%eLL` are assigned to scalar, vector, and matrix variables, respectively. Data of type `%sL` are assumed to be function definitions and are assigned verbatim to the identifier.

In this example, `R` is invoked with an environment containing the declaration of a variable `x` as a scalar equal to 1. The value of $1+1$ is computed by executing the command to add 1 to `x`.

```
$ fun cli --m="ask(R)/<'x': 1.> <'x+1'>" --c %sLm
<'x+1': <'[1] 2'>>
```

octave

This shell interfaces with the GNU Octave system for numerical computation. It allows real valued scalars, vectors, and matrices to be initialized automatically as variables in the interactive environment when used as a parameter to the ask function, from values of type %e, %eL, and %eLL, respectively. It also allows a value of type %sL to be used as a function definition. Because most results from Octave are numerical, the interface specifies a postprocessor that automatically converts the output from character strings to floating point format where applicable.

In this example, octave is used to compute the sum of a short vector of two items.

```
$ fun cli -m="ask(octave)/<'x': <1.,2.>> <'sum(x)'>" -c %em
<'sum(x)': 3.000000e+00>
```

gp

This shell interfaces to the PARI/GP package, which is geared toward high performance numerical and symbolic calculations in exact rational, modular, and arbitrary precision floating point arithmetic, with emphasis on power series. Documentation about this system can be found at http://pari.math.u-bordeaux.fr. Scalar values, vectors, and matrices of strings and all numeric types including arbitrary precision (%E) are recognized and initialized. A list of strings is interpreted as a function definition rather than a vector if the = character appears anywhere within it.

This example asks gp to compute $1 + 1$.

```
$ fun cli --m="(ask gp)/<> <'1+1'>" --c %sLm
<'1+1': <'2'>>
```

scilab

This shell interfaces with the scilab system, which performs numerical calculations with applications to linear algebra and signal processing. Scalars, vectors, and matrices of all numeric types and strings can be recognized and initialized as variables in the workspace when this shell parameterizes the ask function. A list of strings is interpreted as a function definition rather than a vector if the = character appears anywhere in it.

This example asks scilab to compute $1 + 1$.

```
$ fun cli --m="(ask scilab)/<> <'1+1'>" --c %sLm
<'1+1': <'    2.  '>>
```

22.3.3 Computer algebra packages

The interfaces documented in this section pertain to computer algebra packages, which are used primarily for symbolic computations.

gap

This shell interfaces with the `gap` system, which pertains to group theory and abstract algebra, as documented at `http://www.gap-system.org`. Scalars, vectors, and matrices of natural numbers, rational numbers, and strings (but not floating point numbers) can be declared automatically in the workspace when `gap` is used as a parameter to the `ask` function. These are indicated respectively by values of type `%n`, `%nL`, `%nLL`, `%q`, `%qL`, `%qLL`, `%s`, `%sL`, and `%sLL`. However, if any string in a list of strings contains the word "`function`", then the list is treated as a function definition and assigned verbatim to the identifier rather than being initialized as a vector of strings.

This example demonstrates the use of rational numbers with `gap`.

```
$ fun cli --m="ask(gap)/<'x': 1/2> <'x+2/3'>" --c %sLm
<'x+2/3;': <'7/6'>>
```

Most commands to `gap` need to be terminated by a semicolon or else `gap` will wait indefinitely for further input. The shell interface will therefore automatically supply a semicolon where appropriate if it is omitted.

axiom

This shell interfaces with the `axiom` computer algebra system, which is documented at `http://savannah.nongnu.org/projects/axiom`. Scalars, vectors, and matrices of all numeric types and strings are recognized when this shell is the parameter to the `ask` function. A list of strings is treated as a function definition rather than a vector of strings if any string in it contains the = character. Vectors and matrices of strings are declared as symbolic expressions rather than quoted strings.

Any automated driver for the `Axiom` command line interpreter is problematic because the interpreter responds with sequentially numbered prompts that can't be disabled, and the number isn't incremented unless an operation is successful. Errors in commands will therefore cause the client to deadlock rather than raising an exception, as it waits indefinitely for the next prompt in the sequence.

A further difficulty stems from the default two dimensional text output format being impractical to parse for use by another application. However, a partial workaround for this issue is to display an expression x using the type cast x`::INFORM` on the `Axiom` command line, which will cause most expressions to be displayed in `lisp` format. This notation can be transformed to a parse tree by the function `axparse` defined in the `cli` library for this purpose, and documented subsequently in this chapter.

maxima

This shell interfaces to the `Maxima` computer algebra system, as documented at `http://www.sourceforge.net/projects/maxima`. When `maxima` parameterizes the `ask` function, only strings and lists of strings are usable to initialize variables in the workspace (i.e., not vectors or matrices of numeric types as with other interfaces). These are assigned verbatim to their identifiers.

The scripting language for `Maxima` allows interactive routines to be written that prompt the user for input. These should be avoided via this interface because a non-standard prompt will cause the client to deadlock.

22.4 Functions based on shells

A small selection of functions using some of the standard shells is included in the `cli` library for illustrative purposes and possible practical use.

22.4.1 Front ends

The following functions use `bash`, `octave`, or `R` as back ends to compute mathematical results or perform system calls.

now

This function ignores its argument and returns the system time in a character string.

Here is an example of `now`.

```
$ fun cli --m=now0 --c %s
'Sat, 07 Jul 2007 07:07:07 +0100'
```

eigen

This function takes a real symmetric matrix of type `%eLL` to the list of pairs `<(<`$x \ldots$`>,`λ`)`$\ldots$`>` representing its eigenvectors and eigenvalues in order of decreasing magnitude.

Here is an example of the above function.

```
$ fun cli --m="eigen<<2.,1.>,<1.,2.>>" --c %eLeXL
<
   (<7.071068e-01,7.071068e-01>,3.000000e+00),
   (
      <-7.071068e-01,7.071068e-01>,
      1.000000e+00)>
```

A similar result can be obtained with less overhead by the function `dsyevr` among others available through the virtual machine's `lapack` library interface if it is appropriately configured.

`choleski`

This function takes a positive definite matrix of type `%eLL` and returns its lower triangular Choleski factor. If the argument is not positive definite, an exception is raised with a diagnostic message to that effect.

Here are some examples of Choleski decompositions.

```
$ fun cli --m="choleski<<4.,2.>,<1.,8.>>" --c %eLL
<
   <2.000000e+00,0.000000e+00>,
   <1.000000e+00,2.645751e+00>>
$ fun cli --m="choleski<<1.,2.>,<3.,4.>>" --c %eLL
fun:command-line: error: chol: matrix not positive definite
```

The latter example demonstrates the technique of passing through a diagnostic message from the back end `octave` application. Note that if the virtual machine is configured with a `lapack` interface, a quicker and more versatile way to get Choleski factors is by the `dpptrf` and `zpptrf` functions.

`stdmvnorm`

This function takes a triple $(<a_0 \dots a_n>,<b_0 \dots b_n>,\sigma)$ to the probability that a random draw $<x_0 \dots x_n>$ from a multivariate normally distributed population with means 0 and covariance matrix σ has $a_i \leq x_i \leq b_i$ for all $0 \leq i \leq n$.

`mvnorm`

This function takes a quadruple $(<a_0 \dots a_n>,<b_0 \dots b_n>,<\mu_0 \dots \mu_n>,\sigma)$ to the probability that a random draw $<x_0 \dots x_n>$ from a multivariate normally distributed population with means $<\mu_0 \dots \mu_n>$ and covariance matrix σ has $a_i \leq x_i \leq b_i$ for all $0 \leq i \leq n$.

It would be difficult to find a better way of obtaining multivariate normal probabilities than by using the `R` shell interface as these functions do, because there is no corresponding feature in the system's C language API.

22.4.2 Format converters

A couple of functions are usable for transforming the output of a shell. In the case of `Axiom`, the default output format is somewhat difficult to parse.

```
$ fun cli --m="ask(axiom)/<> <'(x+1)^2'>" --c %sLm
```

```
<
   '(x+1)^2': <
      '         2',
      '   (1)  x  + 2x + 1',
      '                            Type: Polynomial Integer'>>
```

Although suitable for interactive use, this format makes for awkward input to any other program. However, the following technique can at least transform it to a `lisp` expression.

```
$ fun cli --m="ask(axiom)/0 <'((x+1)^2)::INFORM'>" --c %sLm
<
   '((x+1)^2)::INFORM': <
      '   (1)  (+ (+ (** x 2) (* 2 x)) 1)',
      '                                        Type: InputForm'>>
```

This format can be made convenient for further processing (e.g., with tree traversal combinators) by the following function.

axparse

Given a `lisp` expression displayed by `Axiom` with an `INFORM` type cast, this function parses it to a tree of character strings.

The following example demonstrates this effect.

```
$ fun cli --c %sT \
> --m="axparse ~&hm ask(axiom)/<> <'((x+1)^2)::INFORM'>"
'+'^: <
   '+'^: <
      '**'^: <'x'^: <>,'2'^: <>>,
      '*'^: <'2'^: <>,'x'^: <>>>,
   '1'^: <>>
```

octhex

This function is used to convert hexadecimal character strings displayed by `Octave` to their floating point representations.

The `octhex` function is used internally by the `octave` interface but may be of use for customizing or hacking it.

```
$ octave -q
octave:1> format hex
octave:2> 1.234567
ans = 3ff3c0c9539b8887
octave:3> quit
$ fun cli --m="octhex '3ff3c0c9539b8887'" --c %e
1.234567e+00
```

22.5 Defining new interfaces

The remainder of the chapter needs to be read only by developers wishing to modify or extend the set of existing shell interfaces. To this end, the basic building blocks are what will be called protocols and clients.

- A protocol is a declarative specification of a prescribed interaction or fragment thereof between a client and a server.
- A client is a virtual machine code program capable of executing a protocol when used as the operand to the virtual machine's `interact` combinator.
- A server in this context is the shell or command line interpreter for which an interface is sought, and is treated as a black box.
- An interface is a record made up of a combination of clients, protocols, or client generating functions each detailing a particular phase of the interaction, such as authentication, initialization, *etcetera*.

22.5.1 Protocols

A protocol is represented as a non-empty list $<(c_0, p_0),\ \ldots (c_n, p_n)>$ of pairs of lists of strings wherein each c_i is a sequence of commands sent by the client to the server, and the corresponding p_i is the text containing the prompt that the server is expected to transmit in reply.

- Line breaks are not explicitly encoded, but are implied if either list contains multiple strings.
- If and when all transactions in the list are completed, the connection is closed by the client and the session is terminated.

Certain patterns have particular meanings in protocol specifications. These interpretations are a consequence of the virtual machine's `interact` combinator semantics.

- If any prompt p_i is a list of one string containing only the end of file character (ISO code 4), the client waits for all output until the server closes the connection and then the session is terminated.
- If a prompt p_i is `<''>`, the list of the empty string, the client waits for no output at all from the server and proceeds immediately to send the next list commands c_{i+1}, if any.
- If a prompt p_i is `<>`, the empty list, the client waits to receive exactly one character from the server and then proceeds with the next command, if any.

The last alternative, although supported by the virtual machine, is not presently used in the `cli` library. It could have applications to matching wild cards in prompts.

The following definitions are supplied in the `cli` library as mnemonic aids in support of the above conventions.

`eof`

the end of file character, ISO code 4, defined as `4%cOi&`

`handshake`

Given a pair $(p,<c_0,\ \dots c_n>)$ where p and c_i are character strings, this function constructs the protocol $\texttt{<(<}c_0\texttt{,'{}'>,<'{}',}p\texttt{>), ... (<}c_n\texttt{,'{}'>,<'{}',}p\texttt{>)>}$ describing a client that sends each command c_i followed by a line break and waits to receive the string p preceded by a line break from the server after each one.

`completing`

Given any protocol $<(c_0,p_0),\ \dots(c_n,p_n)>$, this function constructs the protocol $<(c_0,p_0),\ \dots(c_n,\texttt{<<eof>>})>$, which differs from the original in that the client waits for the server to close the connection after the last command.

`closing`

Given any protocol $<(c_0,p_0),\ \dots(c_n,p_n)>$, this function constructs the protocol $<(c_0,p_0),\ \dots(c_n,\texttt{<'{}'>})>$, which differs from the original in that the connection is closed immediately after the last command without the client waiting for another prompt.

22.5.2 Clients

A client in this context is a function f expressed in virtual machine code that is said to execute a protocol $<(c_0,p_0),\ \dots(c_n,p_n)>$ if it meets the condition

$$\begin{aligned} \forall <x_0\dots x_n>.\ \exists <q_0\dots q_n>.\ f() &= (q_0,c_0,p_0) \\ \wedge\ \forall i\in\{0\dots n-1\}.\ f(q_i,\texttt{-[-[}x_i\texttt{]--[}p_i\texttt{]-]-}) &= (q_{i+1},c_{i+1},p_{i+1}) \end{aligned}$$

where each x_i is a list of character strings and the dash bracket notation has the semantics explained on page 118, in this case concatenating a pair of lists of strings by concatenating the last string in x_i with the first one in p_i, if any. The q_i values are constants of unrestricted type.

A client f in itself is only an alternative representation of a protocol in an intensional form, but when a program `interact` f is applied to any argument, the virtual machine carries out the specified interactions to return the transcript

$$<c_0,\texttt{-[-[}x_0\texttt{]--[}p_0\texttt{]-]-},\dots c_n,\texttt{-[-[}x_n\texttt{]--[}p_n\texttt{]-]-}>$$

with the x values emitted by a server.

The `cli` library contains a small selection of functions for constructing or transforming clients more easily than by hand coding them, which are documented below.

Clients from strings

expect

Given a protocol r, this function returns a client f that executes r in the sense defined above.

exec

Given a single character string s, this function returns a client that is semantically equivalent to expect completing handshake/0 <s>, which is to say that the client specifies the launch of s followed by the collection of all output from it until the server closes the connection.

An example of the above function follows.

```
$ fun cli --m="interact(exec 'uname') 0" --c %sLL
<<'uname'>,<'Linux'>>
```

Clients from clients

seq

This function takes a prompt p to a function that takes a list of clients to their sequential composition in a shell with prompt p. The sequential composition is a client that begins by behaving like the first client in the list, then the second when that one terminates, and so on, expecting the prompt p in between.

- If any client in the list closes the connection, interaction with the next one starts immediately.
- If any client waits for the server to close the connection (with <<eof>>), the prompt <'',p> is expected instead (i.e., p preceded by a line break), any accompanying command from the client has a line break appended, and the interaction of the next client in the list commences when <'',p> is received.
- If the initial output transmitted by any client after the first one in the list is a single string, a line break is appended to the command (by way of an empty string).
- If the initial prompt for any client after the first one in the list is a single string, a line break is inserted at the beginning of the prompt (by way of an empty string).

For a list of commands x and a prompt p, the following equivalence holds,

$$\texttt{expect handshake/}p\,x \equiv \texttt{(seq }p\texttt{) exec*}\ x$$

but the form on the left is more efficient.

Some command line interpreters, such as those of `Axiom` and `Maxima`, use numbered prompts. In these cases, the following function or something similar is useful as a wrapper.

prompt_counter

This function takes a client as an argument and returns a client as a result. For any state in which the given client would expect a prompt containing the substring `'\n'`, the resulting client expects a similar prompt in which this substring is replaced by a natural number in decimal that is equal to 1 for the first interaction and incremented for each subsequent one.

Execution of clients

watch

Given a client as an argument, this function returns a list of type `%scLULL` containing a transcript of the client/server interactions. The function is defined as `~&iNHiF+ interact`.

The `watch` function is a useful diagnostic tool during development of new protocols or clients. Here is an example.

```
$ fun cli --m="watch exec 'ps'" --c %sLL
<
   <'ps'>,
   <
      '  PID TTY          TIME CMD',
      ' 7143 pts/5    00:00:00 ps'>>
```

However, the `watch` function is ineffective if deadlock is a problem, in which case the `--trace` compiler option may be more helpful. See page 281 for an example.

22.5.3 Shell interfaces

The purpose of a `shell` data structure is to encapsulate as much useful information as possible about invoking a shell or command line interpreter. When a `shell` is properly constructed, it can be used as a parameter to the `ask` function and allow easy access to the application it describes. Working with this data structure is explained in this section.

Data structures

As noted below, some of the fields in a `shell` are character strings, but to be adequately expressive, others are protocols, clients, or functions that generate clients, as these terms are understood based on the explanations in the previous sections.

`shell`

This function is the mnemonic for a record with the following fields.

- `opener` – command to invoke the shell, a character string
- `login` – password negotiation protocol, if required, as a list of pairs of lists of strings
- `prompt` – shell prompt to expect, a character string
- `settings` – a list of character strings giving commands to be executed when the shell opens
- `declarer` – a function taking an assignment $(n\colon m)$ to a client that binds the value of m to the symbol n in the shell's environment
- `releaser` – a function taking an assignment $(n\colon m)$ to a client that releases the storage for the symbol n if required; empty otherwise
- `closers` – a list of character strings containg commands to be executed when closing the connection
- `answerer` – a postprocessing function for answers returned by the `ask` function, taking an argument $n\colon m$ of type `%ssLA`, and returning a modified version of m, if applicable
- `nop` – a string containing a shell command that does nothing, used by the `ask` function as a placeholder, usually just the empty string
- `wrapper` – a function used to transform the whole client generated by the `sh` function allowing for anything not covered above

Some additional notes about these fields are given below.

- If the shell has any command line options that are appropriate for non-interactive use, they should be included in the `opener`. e.g., `'R -q'` to launch `R` in "quiet" mode. Any options that disable history, color attributes, banners, and line editing are appropriate.
- The `login` protocol is executed immediately after the `opener`, and should be something like `<(<''>,<'Password: '>),(<'pass',''>,<'$> '>)>` for an application that prompts for a password `pass` and then starts with a prompt `$>`. If no authentication is required, the `login` field can be empty.

- After logging in and executing the first command in the `settings`, the client detects that the server is waiting for more input when a line break followed by the `prompt` string is received. The `prompt` field should therefore contain the whole prompt used by the application from the beginning of the line.
- The argument n: m to the `declarer` and the `releaser` functions comes from the left argument in the expression `(ask` s`)/<`n: m `...>` c when the shell s is used as a parameter to the `ask` function. The functions typically will detect the type of m, and generate a client accordingly of the form `expect completing handshake...` that executes the relevant initialization commands.
  - Most applications have documented or undocumented limits to the maximum line length for interactive input, so initialization of large data structures should be broken across multiple lines.
  - The prompt used by the application during input of continued lines may differ from the main one.
- The `answerer` function, if any, should be envisioned as being implicitly invoked at the point `^(~&n,~answerer` s`)* (ask` s`)/`e c when the shell s is used as a parameter to the `ask` function. Typical uses are to remove non-printing characters or redundant information.
- The `ask` function uses the `nop` command specified in the `shell` data structure as a separator before and after the main command sequence to parse the results. Some applications, such as `Maxima`, do not ignore an empty input line, in which case an innocuous and recognizable command should be chosen as the `nop`.
- Applications with irregular interfaces demanding a hand coded client can be accommodated by the `wrapper` function. The `prompt_counter` function documented in the previous section is one example.

Hierarchical shells

A `shell` data structure can be converted to a client function by the operations listed below. One reason for doing so might be to specify the `declarer` or `releaser` fields in terms of shells, as `bash` does.

`sh`

This function takes an argument of type `_shell` and returns function that takes a pair (e, c) of an environment e and a list of commands c to a client.

`ssh`

Defined as `sh++ hop`, this function takes a pair (h, p) of a host name h and a password p, and returns a function similar to `sh` except that it requires the shell to be executed remotely.

The functions `sh` and `ssh` follow similar calling conventions to `ask` and `sask`, respectively, but return only a client without executing it. Further levels of remote invocation are possible by using the `hop` function explicitly in conjunction with these. Aside from using the client constructed by one of these functions to specify a field in a `shell`, the only useful thing to do with it is to run it by the `watch` function.

```
$ fun cli --m="watch (sh R)/<'x': 1.> <'x+1'>" --c
<
   <'R -q'>,
   <'> '>,
   <'x=1.000000000000000000e+00',''>,
   <'x=1.000000000000000000e+00','> '>,
   <'x+1',''>,
   <'x+1','[1] 2','> '>,
   <'q()',''>,
   <'q()'>>
```

open

This function takes an argument of type `_shell` and returns function that takes a pair (e, c) of an environment e and a list of clients c to a client.

sopen

Defined as `open++ hop`, this function takes a pair (h, p) of a host name and a password, and returns a function similar to `open` except that it requires the shell to be executed remotely.

The functions `open` and `sopen` are analogous to `sh` and `ssh`, except that the operand c is not a list of character strings but a list of clients. The following equivalence holds.

$$\texttt{(sh } s\texttt{)/}e\ c \equiv \texttt{(open } s\texttt{)/}e\ \texttt{exec}\star\ c$$

The `open` function is therefore a generalization of `sh` that provides the means for interactive commands or shells within shells to be specified. It is possible to perform a more general class of interactions with `open` than with the `ask` function, but parsing the transcript into a convenient form (e.g., a list of assignments) must be hand coded.

22.5.4 Interface example

The programming language `yorick` is suitable for numerical applications and scientific data visualization (see `http://yorick.sourceforge.net`), and it is designed to be accessed by a command line interpreter. Although there is no interface to the `yorick` interpreter defined in the `cli` library, a user could easily create one by gleaning the following facts from the documentation.

- The command to invoke the interpreter is `yorick`, with no command line options.
- The interpreter uses the string `'> '` as a prompt, except for continued lines of input, where it uses `'cont> '`.
- The command to end a session is `quit`.
- Two types of objects that can be defined in the environment are floating point numbers and functions.
  - Declarations of floating point numbers use the syntax

    $\langle$*identifier*$\rangle$=$\langle$*value*$\rangle$;

  - Function declarations use the syntax

    func $\langle$*name*$\rangle$ ($\langle$*parameter list*$\rangle$)
    {
      $\langle$*body*$\rangle$
    }

The first three points above indicate the appropriate values for the `opener`, `prompt`, and `closers` fields in the shell specification, while the last point suggests a convenient `declarer` definition. In particular, given an argument n: m, the `declarer` should check whether m is a floating point number or a list of strings. If it is a floating point number, the `declarer` will return a simple client constructed by the `exec` function that performs the assignment in the syntax shown. Otherwise, it will return a client that performs the function declaration by expecting a handshaking protocol with the prompt `'cont> '`.

The complete specification for the shell interface along with a small test driver is shown in Listing 22.1. Assuming this listing is stored in a file named `ytest.fun`, its operation can be verified as follows.

```
$ fun flo cli ytest.fun --show
<'double(x)+1': <'3'>>
```

If this code hadn't worked on the first try, perhaps due to deadlock or a syntax error, the cause of the problem could have been narrowed down by tracing the interaction using the compiler's `--trace` command line option.

```
$ fun flo cli ytest.fun --show --trace
opening yorick
waiting for 62 32

⋮

<- q 113
<- u 117
```

Listing 22.1 example of a user-defined shell interface with a test driver

```
#import std
#import nat
#import cli
#import flo

yorick =

shell[
   opener: 'yorick',
   prompt: '> ',
   declarer: %eI?m(
      ("n","m"). exec "n"--' = '--(printf/'%0.20e' "m")--';',
      %sLI?m(
         expect+ completing+ handshake/'cont> '+ ~&miF,
         <'unknown yorick type'>!%)),
   closers: <'quit'>]

alas =

%sLmP (ask yorick)(
    <
      'x': 1.,
      'double': -[
         func double(x)
            {
               return x+x;
            }]->,
   <'double(x)+1'>)
```

```
<- i 105
<- t 116
<-   10
waiting for 13 10
-> q 113
-> u 117
-> i 105
-> t 116
->   13
->   10
matched
closing yorick
<'double(x)+1': <'3'>>
```

Part IV

Compiler Internals

Yeah well, new rules.

Tom Cruise in *Rain Man*

23 Customization

Many features of Ursala normally considered invariant, such as the operator semantics, can be changed by the command line options listed in Table 23.1. These changes are made without rebuilding or modifying the compiler. Instead, the compiler supplements its internal tables by reading from a binary file whose name is given as a command line parameter. This chapter is concerned with preparing the binary files associated with these options, which entails a knowledge of the compiler's data structures.

The kinds of things that can be done by means explained in this chapter are adding a new operator or directive, changing the operator precedence rules, defining new type constructors and pointers, or even defining new command line options. It is generally assumed that the reader has a reason for wanting to add features to the language, and that the desired enhancements can't be obtained by simpler means (e.g., defining a library function or using programmable directives).

The possible modifications described in this chapter affect only an individual compilation when the relevant command line option is selected, but they can be made the default behavior by editing the compiler's wrapper script. There is likely to be some noticeable overhead incurred when the compiler is launched, which could be avoided if the changes were hard coded. Further documentation to that end is given in the next chapter, but this chapter is worth reading regardless, because the same data structures are involved.

23.1 Pointers

The pointer constructors documented in Chapter 2 are specified in a table called `pnodes` of type `_pnode%m` defined in the file `src/psp.fun`. Each record in the table has the following fields.

- `mnemonic` – either a string of length 1 or a natural number as a unique identifier

option	interpretation
`--help-topics`	load interactive help topics from a file
`--pointers`	load pointer expression semantics from a file
`--precedence`	load operator precedence rules from a file
`--directives`	load directive semantics from a file
`--formulators`	load command line semantics from a file
`--operators`	load operator semantics from a file
`--types`	load type expression semantics from a file

Table 23.1: command line options pertaining to customization

- `pval` – a function taking a tuple of pointers to a pointer
- `fval` – a function taking a tuple of semantic functions to a semantic function
- `pfval` – a function taking a pointer on the left and a semantic function on the right to a semantic function
- `help` – a character string describing the pointer for interactive documentation
- `arity` – the number of operands the pointer constructor requires
- `escaping` – a function taking a natural number escape code to a `_pnode`

Each assignment a: b in the table of `pnodes` has a equal to the `mnemonic` field of b. Hence, we have

```
$ fun psp --m=pnodes --c _pnode%m
<
   'n': pnode[
      mnemonic: 'n',
      pval: 4%fOi&,
      help: 'name in an assignment'],
   'm': pnode[
      mnemonic: 'm',
      pval: 4%fOi&,
      help: 'meaning in an assignment'],
```

⋮

and so on.

The semantics of a given pointer operator or primitive is determined by the fields `pval`, `fval`, and `pfval`. No more than one of them needs to be defined, but it may be useful to define both `pval` and `fval`. The `fval` field specifies a pseudo-pointer semantics, and the `pval` field is for ordinary pointers. The `pfval` field is peculiar to the `P` operator.

Listing 23.1 source file defining a new pseudo-pointer

```
#import std
#import nat
#import psp

#binary+

pfi =

~&iNC pnode[
   mnemonic: 'u',
   fval: ("f","g"). subset^("f","g"),
   arity: 2,
   help: 'binary subset combinator']
```

23.1.1 Pointers with alphabetic mnemonics

An example of a file specifying a new pointer constructor is shown in Listing 23.1. The file contains a list of `pnode` records to be written in binary form to a file named `pfi`. The list contains a single pointer constructor specification with a mnemonic of `u`. This constructor is a pseudo-pointer that requires two pointers or pseudo-pointers as subexpressions in the pointer expression where it occurs. If the expression is of the form `~&`fg`u` x, then the result will be `subset(~&`f x`,~&`g x`)`.

As a demonstration, the text in Listing 23.1 can be saved in a file named `pfi.fun` and compiled as shown.

```
$ fun psp pfi.fun
fun: writing `pfi'
```

Using this file in conjunction with the `--pointers` command line option shows the new pointer is automatically integrated into the interactive help.

```
$ fun --pointers ./pfi --help pointers,2

pointer stack operators of arity 2  (*pseudo-pointer)
-----------------------------------------------------
   A    assignment constructor
⋮
 * p    zip function
 * u    binary subset combinator
 * w    membership
```

As this output shows, the rest of the pointers in the language retain their original meanings when a new one is defined, and the new ones replace any built in pointers having the same mnemonics. Another alternative is to use the `only` parameter on the command line, which will make the new pointers the only ones that exist in the language.

```
$ fun --main="~&x" --decompile
main = reverse
$ fun --pointers only ./pfi --main="~&x" --decompile
fun:command-line: unrecognized identifier: x
```

A simple test of the new pointer is the following.

```
$ fun --pointers ./pfi --m="~&u/'ab' 'abc'" --c %b
true
```

A more reassuring demonstration may be to inspect the code generated for the expression `~&u`, to confirm that it computes the subset predicate.

```
$ fun --pointers ./pfi --m="~&u" --d
main = compose(
   refer conditional(
      field(0,&),
      conditional(
         compose(member,field(0,(((0,&),(&,0)),0))),
         recur((&,0),(0,(0,&))),
         constant 0),
      constant &),
   compose(distribute,field((0,&),(&,0))))
```

23.1.2 Pointers accessed by escape codes

A drawback of defining a new pointer in the manner described above is that the mnemonic `u` is already used for something else. Although it is easy to change the meaning of an existing pointer, doing so breaks backward compatibility and makes the compiler unable to bootstrap itself. The issue is not avoided by using a different mnemonic because every upper and lower case letter of the alphabet is used, digits have special meanings, and non-alphanumeric characters are not valid in pointer mnemonics. However, it is possible to define new pointer operators by using numerical escape codes as described in this section.

The `escaping` field in a `pnode` record may contain a function that takes a natural number as an argument and returns a `pnode` record as a result. The argument to the function is derived from the digits that follow the occurrence of the escaping pointer in an expression. The result returned by the `escaping` field is substituted for the original and the escape code to evaluate the expression.

There is only one pointer in the `pnodes` table that has a non-empty `escaping` field, which is the `K` pointer, but only one is needed because it can take an unlimited number of escape codes. The way of adding a new pointer as an escape code is to redefine the `K` pointer similarly to the previous section, but with the `escaping` field amended to include the new pointer.

A simple way of proceeding is to use the definitions of the `K` pointer and the `escapes` list from the `psp` module, as shown in Listing 23.2. The `escapes` list is a list of type

Listing 23.2 adding a new pointer without breaking backward compatibility

```
#import std
#import nat
#import psp

pfi =

~&iNC pnode[
   mnemonic: length psp-escapes,
   fval: ("f","g"). subset^("f","g"),
   arity: 2,
   help: 'binary subset combinator']

escapes = --(^A(~mnemonic,~&)* pfi) psp-escapes

#binary+

kde =

~&iNC pnode[
   mnemonic: 'K',
   fval: <'escape code missing after K'>!%,
   help: 'escape to numerically coded operators',
   escaping: %nI?(
      ~&ihrPB+ ^E(~&l,~&r.mnemonic)*~+ ~&D\(~&mS escapes),
      <'numeric escape code missing after K'>!%),
   arity: 1]
```

_pnode%m whose i-th item (starting from 0) has a mnemonic equal to the natural number i. It is used in the definition of the escaping field of the K pointer specification.

The K record is cut and pasted from psp.fun, without any source code changes, but the list of escapes is locally redefined to have an additional record appended. Appending it rather than inserting it at the beginning is necessary to avoid changing any of the existing escape codes. The appended record, for the sake of a demonstration, is similar to the one defined in the previous section.

The code in Listing 23.2 is compiled as shown.

```
$ fun psp kde.fun
fun: writing `kde'
```

The new pointer shows up as an escape code as required in the interactive help,

```
$ fun --pointers ./kde --help pointers,2

pointer stack operators of arity 2  (*pseudo-pointer)
------------------------------------------------------
⋮
 * K18  binary subset combinator
⋮
```

and it has the specified semantics.

```
$ fun --pointers ./kde --m="~&K18" --d
main = compose(
   refer conditional(
      field(0,&),
      conditional(
         compose(member,field(0,(((0,&),(&,0)),0))),
         recur((&,0),(0,(0,&))),
         constant 0),
      constant &),
   compose(distribute,field((0,&),(&,0))))
```

23.2 Precedence rules

The --precedence command line option allows the operator precedence rules documented in Section 5.1.3 to be changed. The option requires the name of a binary file to be given as a parameter, that contains a pair of pairs of lists of pairs of strings

$$((\langle \textit{prefix-infix} \rangle, \langle \textit{prefix-postfix} \rangle), (\langle \textit{infix-postfix} \rangle, \langle \textit{infix-infix} \rangle))$$

of type %sWLWW. Each component of the quadruple pertains to the precedence for a particular combination of operators arities (e.g., prefix and infix). Each string is an operator

Listing 23.3 a revised set of precedence rules to make infix composition right associative

```
#binary+

npr = ((<>,<>),(<>,<('+','+')>))
```

mnemonic, either from Table 5.2 or user defined. The presence of a pair of strings in a component of the tuple indicates that the left operator is related to the right under the precedence relation.

23.2.1 Adding a rule

Listing 23.3 provides a short example of a change in the precedence rules. Normally infix composition is left associative, but this specification makes the + operator related to itself when used in the infix arity, and therefore right associative. Given this code in a file named `npr.fun`, we have

```
$ fun --main="f+g+h" --parse
main = (f+g)+h
$ fun npr.fun
fun: writing `npr'
$ fun --precedence ./npr --main="f+g+h" --parse
main = f+(g+h)
```

In the case of functional composition, both interpretations are of course semantically equivalent.

23.2.2 Removing a rule

Additional precedence relationships are easy to add in this way, but removing one is slightly less so. In this case, a set of precedence rules derived from the default precedence rules from the module `src/pru.avm` has to be constructed as shown below, with the undesired rules removed.

```
npr = (&rr:= ~&j\<(';','/')>+ ~&rr) pru-default_rules
```

The rules would then be imposed using the `only` parameter to the `--precedence` option, as in

```
$ fun --precedence only ./npr foobar.fun
```

23.2.3 Maintaining compatibility

Changing the precedence rules can almost be guaranteed break backward compatibility and make the compiler unable to bootstrap itself. If customized precedence rules are implemented after a project is underway, it may be helpful to identify the points of incompatibility by a test such as the following.

```
$ fun *.fun --parse all > old.txt
$ fun --precedence ./npr *.fun --parse all > new.txt
$ diff old.txt new.txt
```

Assuming the files of interest are in the current directory and named `*.fun`, this test will identify all the expressions that are parsed differently under the new rules and therefore in need of manual editing.

23.3 Type constructors

Type expressions are represented as trees of records whose declaration can be found in the file `src/tag.fun`. The main table of type constructor records is declared in the file `src/tco.fun`. It has a type of `_type_constructor%m`. A `type_constructor` record has the following fields, first outlined briefly below and then explained in more detail.

- `mnemonic` – a string of exactly one character uniquely identifying the type constructor
- `microcode` – a function that maps a pair (s, t) with a stack of previous results s and a list of type constructors t to a new configuration (s', t')
- `printer` – given a pair `(<`$t\ldots$`>,`x`)`, where <$t\ldots$> is a stack of type expressions and x is an instance, the function in this field returns a list of character strings displaying x as an instance of type t. Trailing members of <$t\ldots$>, if any, are the ancestors of t in the expression tree were it occurs.
- `reader` – for some primitive types, this field contains an optional function taking a list of character strings to an instance of the type
- `recognizer` – same calling convention as the `printer`, returns true iff x is an instance of the type t
- `precognizer` – same as the recognizer except without checking for initialization
- `initializer` – a function taking an argument of the form `(<`$f\ldots$`>,<`$t\ldots$`>)` where <$t\ldots$> is a stack of type expressions as above, and <$f\ldots$> is a list of type initializing functions with one for each subexpression; the result is the main initialization function for the type
- `help` – short character string to be displayed by the compiler for interactive help
- `arity` – natural number specifying the number of subexpressions required
- `target` – used by the `microcode` to store a function value
- `generator` – takes a list <$g\ldots$> of one generating function for each subexpression and returns random instance generator for the whole type expression

23.3.1 Type constructor usage

Supplementary material on the `type_constructor` field interpretations is provided in this section for readers wishing to extend or modify the system of types in the language. As noted above, every field in the record except for the `help` and `arity` fields is a function. Most of these functions are not useful by themselves, but are intended to be combined in the course of a traversal of a tree of type constructors representing an aggregate type or type related function. This design style allows arbitrarily complex types to be specified in terms of interchangeable parts, but it requires the functions to follow well defined calling conventions.

Printer and recognizer calling conventions

The printing function for a type d`^: `v, where d is a `type_constructor` record, is computed according to the equivalence

$$(\texttt{\%-P }\ d\texttt{\^{}: }\ v)\ x \equiv (\texttt{\~{}printer }\ d)\ (<d\texttt{\^{}: }\ v\texttt{>,}x)$$

at the root level. Note that the function is applied to an argument containing itself and the type expression in which it occurs, which is convenient in certain situations, in addition to the data x to be printed.

Primitive and aggregate type printers For primitive types, the `printer` field often may take the form f`+ ~&r`, because the type expressions on the left are disregarded. For example, the printer for boolean types is as follows.

```
$ fun tag --m="~&d.printer %b" --d
main = couple(
   conditional(
      field(0,&),
      constant 'true',
      constant 'false'),
   constant 0)
```

For aggregate types, the `printer` in the root constructor normally needs to invoke the printers from the subexpressions at some point. When a printer for a subexpression is called, convention requires it to be passed an argument of the form

$$(\texttt{<}t, d\texttt{\^{}: }\ v\texttt{>,}x')$$

where d`^: `v is the original type expression, now appearing second in the list, while t is the subexpression type. In this way, the subexpression printer may access not just its own type expression but its parents. Although most printers do not depend on the parents of the expression where they occur, the exception is the `h` type constructor for recursive types (and indirectly for recursively defined records).

List printer example To make this description more precise, we can consider the printer for the list type constructor, `L`. The representation for a list type expression is always something similar to the following,

```
$ fun tag --m="%bL" --c _type_constructor%T
^: (
   type_constructor[
      mnemonic: 'L',
      printer: 674%fOi&,
      recognizer: 274%fOi&,
      precognizer: 100%fOi&,
      initializer: 32%fOi&,
      generator: 1605%fOi&],
   <
      ^:<> type_constructor[
         mnemonic: 'b',
         printer: 80%fOi&,
         recognizer: 16%fOi&,
         initializer: 11%fOi&,
         generator: 110%fOi&]>)
```

where the subexpression may vary. The source code for the `printer` function in the list type constructor takes the form

`^D(~&lhvh2iC,~&r);  (* ^H/~&lhd.printer ~&); ` f

where the function f takes a list of lists of strings to a list of strings, supplying the necessary indentation, delimiting commas, and enclosing angle brackets. The first phase, `^D(~&lhvh2iC,~&r)`, takes an argument of the form

$$(\texttt{<}d\texttt{^:<}t\texttt{>>},\texttt{<}x_0 \dots x_n\texttt{>})$$

and transforms it to a list of the form

$$\texttt{<}(\texttt{<}t, d\texttt{^:<}t\texttt{>>}, x_0) \dots (\texttt{<}t, d\texttt{^:<}t\texttt{>>}, x_n)\texttt{>}$$

The second phase, `(* ^H/~&lhd.printer ~&)`, uses the printer of the subexpression t to print each item x_0 through x_n. Many printers for unary type constructors have a similar first phase of pushing the subexpression onto the stack, but this second phase is more specific to lists.

Recognizers The calling conventions for `recognizer` and `precognizer` functions follow immediately from the one for printers. Rather than returning a list of strings, these functions return boolean values. The root printer function of a type expression may need to invoke the recognizer functions of its subexpressions, which is done for example in the case of free unions.

The difference between the `recognizer` and the `precognizer` field is that the `precognizer` will recognize an instance that has not been initialized, such as a rational number that is not in lowest terms or a record whose initializing function has not been applied. For some types (mainly those that don't have an initializer), there is no distinction and the `precognizer` field need not be specified. However, if the distinction exists, then the `precognizer` needs to reflect it in order for unions and a-trees to work correctly with the type.

Microcode and target conventions

The function in the `microcode` field is invoked when a type expression is evaluated as described in Section 4.3.1. To evaluate an expression such as $s\%t_0t_1\dots t_n$, the list of type constructors $<T_0\dots T_n>$ associated with each of the mnemonics t_0 through t_n is combined with the initial stack $<s>$, and the `microcode` field of T_0 is applied to $(<s>, <T_0\dots T_n>)$. Certain conventions are followed by microde functions although they are not enforced in any way.

- If T_0 is the type constructor for a primitive type, the microcode should return a result of $(<T_0\text{\^{}:}<>, s>, <T_1\dots T_n>)$, which has the unit tree of the constructor T_0 shifted to the stack.
- If T_1 is a unary type constructor, its microcode should map the result returned by the microcode of T_0 to $(<T_1\text{\^{}:}<T_0\text{\^{}:}<>>, s>, <T_2\dots T_n>)$, which shifts a type expression onto the stack having T_1 as the root and the previous top of the stack as the subexpression.
- If T_1 is a binary type constructor, its microcode should map the result returned by the microcode of T_0 to $(<T_1\text{\^{}:}<s, T_0\text{\^{}:}<>>>, <T_2\dots T_n>)$, and s best be a type expression. This result has a type expression on top of the stack with T_1 as the root and the two previous top items as the subexpressions.
- If any T_i represents a functional combinator rather than a type constructor (for example, like the `P` and `I` constructors), the `microcode` should return a result of the form $(<d\text{\^{}:}<>>, <>)$, with the resulting function stored in the `target` field of d.
- The microcode for the remaining constructors such as `l` and `r` transforms the stack in arbitrary *ad hoc* ways, as shown in Figure 4.1 on page 167.

Initializers

The `initializer` field in each type constructor is responsible for assigning the default value of an instance of a type when it is used as a field in a record. It takes an argument of the form $(<f_0\dots f_n>, <t\dots>)$ because the initializer of an aggregate type is normally defined in terms of the initializers of its component types, although the initializer of a primitive type is constant. For example, the boolean (`%b`) initializer is `! ~&i&& &!`, the constant function returning the function that maps any non-empty value to the `true` boolean value (`&`). The initializer of the list construtor (`L`) is `~&l; ~&ihB&& ~&h; *`, the function that applies the initializer f_0, in the above expression, to every item of a list.

For aggregate types, most initializers are of the form `~&l;` h, because they depend only on the initializers of the subtypes, but the exception is the `U` type constructor, whose initializer needs to invoke the `precognizer` functions of its subtypes and hence requires the stack of ancestor types in case any of them is recursively defined.

Generators

A random instance generator for a type t is a function that takes either a natural number as an argument or the constant `&`. If it is given a natural number n as an argument, its job is to return an instance of t having a weight as close as possible to n, measured in quits. If it is given `&` as an argument, it is expected to return a boolean value which is true if there exists an upper bound on the size of the instances of t, and false otherwise. Examples of the former types are boolean, character, standard floating point types, and tuples thereof.

The `generator` field in each type constructor is responsible for constructing a random instance generator of the type. For aggregate types, it is normally defined in terms of the generators of the component types, but for primitive types it is invariant. For example, the `generator` field of the `e` type constructor is defined as

```
! math..sub\10.0+ mtwist..u_cont+ 20.0!
```

whereas the generator of the `U` type constructor is

```
&?=^\choice !+ ~&g+ ~&iNNXH+ gang
```

based on the assumption that it will be applied to the list of the generators of the component types, `<`$g_0 \dots g_n$`>`. Note that `~&g ~&iNNXH gang<`$g_0 \dots g_n$`>` is equivalent to `~&g <.`$g_0 \dots g_n$`> &`, which is non-empty if and only if g_i `&` is non-empty for all i.

Various functions defined in the `tag` module may be helpful for constructing random instance generators, but there are no plans to maintain a documented stable API for this purpose.

23.3.2 User defined primitive type example

Interval arithmetic is a technique for coping with uncertainty in numerical data by identifying an approximate real number with its known upper and lower bounds. By treating the pair of bounds as a unit, sums, differences, and products of intervals can all be defined in the obvious ways.

Interval representation

A library of interval arithmetic operations is beyond the scope of this example, but the specification of a primitive type for intervals is shown in Listing 23.4. According to this specification, intervals are represented as pairs (a, b) with $a < b$, where a and b are

Listing 23.4 a new primitive type for interval arithmetic

```
#import std
#import nat
#import tag
#import flo

#binary+

H =

~&iNC type_constructor[
   mnemonic: 'H',
   microcode: ~&rhPNVlCrtPX,
   printer: ~&r; ~&iNC+ math..isinfinite?l(
      math..isinfinite?r('0+-inf'!,--'-inf'+ ~&h+ %eP+ ~&r),
      math..isinfinite?r(
         --'+inf'+ ~&h+ %eP+ ~&l,
         ^|T(~&,'+-'--)+ (~&h+ %eP+ div\2.)^~/plus bus)),
   reader: ~&L; -?
      (=='0+-inf'): (ninf,inf)!,
      substring/'+-': -+
         math..strtod~~; ~&rllXG; ^|/bus plus,
         (`+,`-)^?=ahthPX/~&Natt2X ~&ahPfatPRXlrlPCrrPX+-,
      suffix/'-inf': ~&/ninf+ math..strtod+ ~&xttttx,
      suffix/'+inf': ~&\inf+ math..strtod+ ~&xttttx,
      <'bad interval'>!%?-,
   recognizer: ! ~&i&& &&fleq both %eI,
   precognizer: ! ~&i&& both %eI,
   initializer: ! ~&?\(ninf,inf)! ~&l?(
      ~&r?/(fleq?/~& ~&rlX) ~&\inf+ ~&l,
      ~&/ninf!+ ~&r),
   help: 'push primitive interval type',
   generator: ! &?=/&! fleq?(~&,~&rlX)+ 0%eWi]
```

floating point numbers representing the endpoints. This representation is implied by the `recognizer` function, which is satisfied only by a pair of floating point numbers with the left less than the right.

Interval type features

The mnemonic for the interval type is `H`, so it may be used in type expressions like `%H` or `%HL`, *etcetera*. This mnemonic is chosen so as not to clash with any already defined, thereby maintaining backward compatibility. A small number of unused type mnemonics is available, which can be listed as shown.

```
$ fun tco --m="~&lrnSL2j/letters type_constructors" --c
'FHK'
```

Other fields in the type constructor are defined to make working with intervals convenient. The `initializer` function will take a partially initialized interval and define the rest of it. If either endpoint is missing, infinity is inferred, and if the endpoints are out of order, they are interchanged. The default value of an interval is the entire real line. This function would be invoked whenever a field in a record is declared as type `%H`.

The `precognizer` field differs from the `recognizer` by admitting either order of the endpoints. This difference is in keeping with its intended meaning as the recognizer of data in a non-canonical form, where this concept applies.

The concrete syntax for a primitive type needn't follow the representation exactly. The `printer` and `reader` fields accommodate a concrete syntax like

```
1.269215e+00+-9.170847e-01
```

for finite intervals, which is meant to resemble the standard notation $x \pm d$ with x at the center of the interval and d as half of its width. Semi-infinite intervals are expressed as x`+inf` or x`-inf` as the case may be, with the finite endpoint displayed.

The `generator` function simply generates an ordered pair of floating point numbers. The size (in quits) of a pair of floating point numbers is not adjustable, so the generator returns `&` when applied to a value of `&`, following the convention.

Interval type demonstration

To test this example, we first store Listing 23.4 in a file named `ty.fun` and compile it as follows.

```
$ fun tag flo ty.fun
fun: writing `H'
```

Random instances can now be generated as shown.

```
$ fun --types ./H --m="0%Hi&" --c %H
-7.577923e+00+-3.819156e-01
```

Note that if the file name `H` doesn't contain a period, it should be indicated as shown on the command line to distinguish it from an optional parameter. Data can also be cast to this type and displayed,

```
$ fun --types ./v --m="(1.6,1.7)" --c %H
1.650000e+00+-5.000000e-02
```

and data using the concrete syntax chosen above can be read by the interval parser `%Hp`.

```
$ fun --types ./H --m="%Hp -[2.5+-.001]-" --c %H
2.500000e+00+-1.000000e-03
```

However, defining a concrete syntax for constants of a new primitive type does not automatically enable the compiler to parse them.

```
$ fun --types ./H --m="2.5+-.001" --c %H
fun:command-line: unbalanced +-
```

This kind of modification to the language would require hand written adjustments to the lexical analyzer, as outlined in the next chapter.

23.4 Directives

The compiler directives, as documented in Chapter 7, are defined in terms of transformations on the compiler's run-time data structures. They can be used either to generate output files or to make arbitrary source level changes during compilation, and in either case may be parameterized or not.

The directive specifications are stored in a table named `default_directives` defined in the file `src/dir.fun`. This table can be modified dynamically when the compiler is invoked with the `--directives` command line option. This option requires a binary file containing a list of directive specifications that will be incorporated into the table. A directive specification is given by a record with the following fields, which are explained in detail in the remainder of this section.

- `mnemonic` – the identifier used for the directive in the source code
- `parameterized` – character string briefly documenting the parameter if one is required
- `parameter` – default parameter value; empty means there is none
- `nestable` – boolean value implying the directive is required to appear in matched + and − pairs (currently true of only the `hide` directive)
- `blockable` – boolean value implying the scope of the directive doesn't automatically extend inside nestable directives (currently true only of the `export` directive)
- `commentable` – boolean value indicationg that output files generated by the directive can have comments included by the `comment` directive

- `mergeable` – boolean value implying that multiple output file generating instances of the directive in the same source file should have their output files merged into one
- `direction` – a function from parse trees to parse trees that does most of the work of the directive
- `compilation` – for output generating directives, a function taking a module and a list of files (type `_file%LomwX`) to a list of files (type `_file%L`)
- `favorite` – a natural number such that higher values cause the directive to take precedence in command line disambiguation
- `help` – a one line description of the directive for on-line documentation

23.4.1 Directive settings

The settings for fields in a `directive` record tend follow certain conventions that are summarized below, and should be taken into account when defining a new directive.

Flags

- The `nestable` and `blockable` fields should normally be false in a directive specification, unless the directive is intended as a replacement for the `hide` or `export` directives, respectively.
- The `commentable` field should normally be true for output generating directives that generate binary files, but probably not for other kinds of files.
- Either setting of the `mergeable` field could be reasonable depending on the nature of the directive. Currently it is true only of the `library` directive.

Command line settings

Any new directive that is defined will automatically cause a command line option of the same name to be defined that performs the same function, unless there is already a command line option by that name, or the directive is defined with a true value for the `nestable` field.

- A non-zero value for the `favorite` may be chosen if the directive is likely to be more frequently used from the command line than existing command line options starting with the same letter. Several directives currently use low numbers like `1`, `2`, *etcetera* (page 277). Higher numbers indicate higher name clash resolution priority.
- The `parameter` field, which can have any type, is not used when the directive occurs in a source file, but will supply a default parameter for command line usage. For example, the `#cast` directive has a `%g` type expression as its default parameter.
- The `help` and `parameterized` fields should be assigned short, meaningful, helpful character strings because these will serve as on-line documentation.

23.4.2 Output generating functions

The remaining fields in a `directive` record describe the operations that the directive performs as functions. The more straightforward case is that of the `compilation` field, which is used only in output generating directives.

Calling conventions

The `compilation` field takes an argument of the form

$$(\texttt{<}s_0 : x_0 \ldots s_n : x_n\texttt{>}, \texttt{<}f_0 \ldots f_m\texttt{>})$$

where s_i is a string, x_i is a value of any type, and f_j is a file specification of type `_file`, as defined in the standard library. These values come from the declarations that appear within the scope of the directive being defined. For example, a user defined directive by the name of `foobar` used in a source file such as the following

```
#foobar+

s = 1.2
t = (3,4.0E5)

#foobar-
```

can be expected to have a value of `(<'s': 1.2,'t': (3,4.0E5)>,<>)` passed to the function in its `compilation` field. Note that the right hand sides of the declarations are already evaluated at that stage. The list of files on the right hand side is empty in this case, but for the code fragment below it would contain a file.

```
#foobar+

s = 1.2
t = (3,4.0E5)

#binary+

u = 'game over'

#binary-

#foobar-
```

The files in the right hand side of the argument to the `compilation` function are those that are generated by any output generating directives within its scope. These files can either be ignored by the function, or new files derived from them can be returned.

Listing 23.5 simple example of an output generating directive

```
directive[
   mnemonic: 'binary',
   commentable: &,
   compilation: ~&l; * file$[
      stamp: &!,
      path: ~&nNC,
      preamble: &!,
      contents: ~&m],
   help: 'dump each symbol in the current scope to a binary file']
```

Example

The resulting list of files returned by the `compilation` function can depend on these parameters in arbitrary ways. Listing 23.5 shows the complete specification for the `binary` directive, whose `compilation` field makes a binary file for each item of the list of declarations.

23.4.3 Source transformation functions

The `direction` field in a `directive` specification can perform an arbitrary source level transformation on the parse trees that are created during compilation. Unlike the `compilation` field, this function is invoked at an earlier stage when the expressions might not be fully evaluated.

Parse trees

Parse trees are represented as trees of `token` records, which are declared in the file `src/lag.fun`. Functions stored in these records allow parse trees to be self-organizing. A bit of a digression is needed at this point to explain them in adequate detail, but this material is also relevant to user defined operators documented subsequently in this chapter. A `token` record contains the following fields.

- `lexeme` – a character string identifying the token as it appears in a source file
- `filename` – a character string containing the name of the file in which the token appears
- `filenumber` – a natural number indicating the position of the token's source file in the command line
- `location` – a pair of natural numbers giving the line and column of the token in its source file
- `preprocessor` – a function whereby the parse tree rooted with this token is to be transformed prior to evaluation

- `postprocessors` – a list of functions whose head transforms the value of the parse tree rooted with this token after evaluation
- `semantics` – a function taking the token's suffix to a function that takes the list of subtrees to the value of the whole tree rooted on this token
- `suffix` – the suffix list (type `%om`) associated with this token in the source file
- `exclusions` – a predicate on character strings used by the lexical analyzer to qualify suffix recognition
- `previous` – an ignored field available for any future purpose

The first four fields are used for name clash resolution as explained on page 253, and the semantic information is contained in the remaining fields. All of these fields except possibly the `semantics` will have been filled in automatically prior to any user defined directive being able to access them.

Control flow during compilation When the compiler is invoked, the first phase of its operation after interpreting its command line options is to build a tree of `token` records containing all of the declarations and directives in all of the source files. Symbolic names appearing in expressions are initially represented as terminal nodes with the `semantics` field undefined, but literal constants have their `semantics` initialized accordingly. This tree is then transformed under instructions contained in the tree itself. The transformation proceeds generally according to these steps.

1. Traverse the tree repeatedly from the top down, executing the `preprocessor` field in each node until a fixed point is reached.
2. Traverse the tree from the bottom up, evaluating any subtree in which all nodes have a known semantics, and replace such subtrees with a single node.
3. Search the tree for subtrees corresponding to fully evaluated declarations, and substitute the values for the identifiers elsewhere in the tree according to the rules of scope.

Control returns repeatedly to the first step after the third until a fixed point is reached, because further progress may be enabled by the substitutions. Hence, there may be some temporal overlap between evaluation and preprocessing in different parts of the tree, rather than a clear separation of phases.

Parse tree semantics Almost any desired effect can be achieved by a directive through suitable adjustment to the `preprocessor`, `postprocessors`, and `semantics` fields of the parse tree nodes, so it is worth understanding their exact calling conventions. The `preprocessor` field is invoked essentially as follows.

```
^= ~&a^& ^aadPfavPMVB/~&f ^H\~&a ||~&! ~&ad.preprocessor
```

Hence, its argument is the tree in whose root it resides, and it is expected to return the whole tree after transformation. The `semantics` field is invoked as if the following code were executed during parse tree evaluation.

```
~&a^& ^H(
   ||~&! ~&ad.postprocessors.&ihB,
   ^H\~&favPM ~&H+ ~&ad.(semantics,lag-suffix))
```

The argument of the `semantics` function is the `suffix` of the node in which it resides. It is expected to return a function that will map the list of values of the subtrees to a value for the whole tree, which is passed to the head of the `postprocessors`, if any, to obtain the final value.

Transformation calling conventions

When a user defined directive has a non-empty `direction` field, this field should contain a function that takes a tree of `token` records as described above and return one that is transformed as desired. The tree represents the source code encompassing the scope of the directive (i.e., everything following it up to the end of the enclosing name space or the point where it is switched off).

The `direction` function benefits from a reflective interface in that the root of the tree passed to it is a `token` whose `lexeme` is the directive's mnemonic and whose `preprocessor` and `semantics` are automatically derived from the `direction` and `compilation` functions of the directive.

For parameterized directives, the parameter is accessed as the first subexpression of the parse tree, `~&vh`. If the action of the directive depends on the value of the parameter, as it typically would, then the parameter needs to be evaluated first. The `direction` function can wait until the parameter is evaluated before proceeding if it is specified in the following form,

```
(*^0 -&~&,~&d.semantics,~&vig&-)?vh\~& f
```

where f is the function that is applied after the parameter has been evaluated. This code simply traverses the first subexpression tree to establish that all `semantics` fields are initialized. If this condition is not met, it means there are symbolic names in the expression that have not yet been resolved, but will be on a subsequent iteration, as explained above in the discussion of control flow. In this case, the identity function `~&` leaves the tree unaltered.

A general point to note about `direction` functions is that some provision usually needs to made to ensure termination when they are iterated. The simplest approach for the directive to delete itself from the tree by replacing the root with a placeholder such as the `separation` token defined in the `apt` library. Where this is not appropriate, it also suffices to delete the `preprocessor` field of the root token. Refer to the file `src/dir.fun` for examples.

Listing 23.6 an example of a directive performing a parse tree transformation

```
#import std
#import nat
#import lag
#import dir
#import apt

#binary+

al =

~&iNC directive[
   mnemonic: 'alphabet',
   direction: _token%TMk+ ~&v?(
      ~&V/separation+ ^T\~&vt -+
         * ~&ar^& ^V\~&falrvPDPM :=ard (
            &ard.(filename,filenumber,location),
            ~&al.(filename,filenumber,location)),
         ^D/~&d ~&vh; -+
            * -+
               ~&V/token[lexeme: '=',semantics: ~&hthPA!],
               ~&iNViiNCC+ token$[lexeme: ~&,semantics: !+ !]+-,
            *^0 ^T\~&vL ~&d.lexeme; &&~&iNC subset\letters+-+-,
      <'misused #alphabet directive'>!%),
   help: 'bulk declare a list of identifiers as strings',
   parameterized: 'list-of-identifiers']
```

23.4.4 User defined directive example

One reason for customizing the directives might be to implement syntactic sugar for some sort of domain specific language. In a language concerned primarily with modelling or simulation of automata, for example, it might be convenient to declare a system's input or output alphabet in an abstract style such as the following.

```
#alphabet <a,b,ack,nack,foo,bar>

system = box_of(a,b,ack,nack)
```

The intent is to allow the symbols `a`, `b`, *etcetera* to be used as symbolic names with no further declarations required.

Specification

Listing 23.6 shows a possible specification for a directive to accomplish this effect, which works by declaring each symbol as a string containing its identifier, (e.g., `a = 'a'`) but this representation need not be transparent to the user. This example could also serve as a

Listing 23.7 test driver for the directive defined in Listing 23.6

```
#alphabet foo bar baz

x = <foo,bar,baz>
```

prototype for more sophisticated alternatives. Several points of interest about this example are the following.

- The parameter to the directive need not be a list of identifiers, but can be any expression the compiler is able to parse. The directive traverses its parse tree in search of alphabetic identifiers and ignores the rest.
- The declaration subtree constructed for each identifier has = as the root token, which is a requirement for a declaration, as is its semantics of `~&hthPA!`, the function that constructs an assignment from the two subexpressions.
- The `semantics` field constructed for each identifier is a second order function of the form x`!!` to follow the convention of returning a function when applied to the suffix (unused in this case) that returns a value when applied to the list of subexpression values (empty in this case).
- The `location` and related fields for the newly created parse trees are inherited from those of the root token of the parse tree to ensure that name clash resolution will work correctly for these identifiers if required.
- The transformation calls for the directive to delete itself from the parse tree so that it won't be done repeatedly. The replacement of the root with the `separation` token accomplishes this effect.

Demonstration

To demonstrate this example, we can store it in a file named `al.fun` and compile it as follows.

```
$ fun lag dir apt al.fun
fun: writing `al'
```

It can then be tested in a file such as the one shown in Listing 23.7, named `altoid.fun`.

```
$ fun --directives ./al altoid.fun --c
<'foo','bar','baz'>
```

This output is what should be expected if the identifiers were declared as strings. We can also verify that the directive is accessible directly from the command line.

```
$ fun --dir ./al --m=foo --alphabet foo --c
'foo'
```

23.5 Operators

The operators documented in Chapters 5 and 6 are specified by a table of records of type `_operator`. The record declaration is in the file `src/ogl.fun`. The main operator table is defined in the file `ops.fun`, the declaration operators are defined in the file `eto.fun`, and the invisible operators for function application, separation, and juxtaposition are defined in the file `apt.fun`.

Adding a new operator to the language or changing the semantics of an existing one is a matter of putting a new record in the table. It can be done dynamically by the `--operators` command line option, which takes a binary file containing a list of operators in the form of `operator` record specifications.

23.5.1 Specifications

Most operators admit more than one arity but have common or similar features that are independent of the arity. The `operator` record therefore contains several fields of type `_mode`. A `mode` record is used as a generic container having a named field for each arity. The field identifiers are `prefix`, `postfix`, `infix`, `solo`, and `aggregate`. This record type is declared in the file `ogl.fun`. Here is a summary of the fields in an `operator` record.

- `mnemonic` – a string of one or two characters containing the symbol used for the operator in source code
- `match` – for aggregate operators, a character string containing the right matching member of the pair (e.g. a closing parenthesis or brace)
- `meanings` – a `mode` of functions containing semantic specifications
- `help` – a `mode` of character strings each being a one line descriptions of the operator for on-line help
- `preprocessors` – a `mode` of optional functions containing additional transformations for the `preprocessor` field in the operator `token`
- `optimizers` – a `mode` of functions containing optional code optimizations or other postprocessing semantics applicable only for compile time evaluation
- `excluder` – an optional predicates taking a character string and returning a true value if it should not be interpreted as a suffix during lexical analysis
- `options` – a module (type `%om`) of entities to be recognized during lexical analysis if they appear in the suffix of the operator
- `opthelp` – a list of strings containing free form documentation of the operator's suffixes as given by the `options` field
- `dyadic` – a `mode` of boolean values indicating the arities for which the dyadic algebraic property holds

- tight – a boolean value indicating higher than normal operator precedence (used by the parser generator)
- loose – a boolean value indicating lower than normal precedence (used by the parser generator)
- peer – an optional mnemonic of another operator having the same precedence (used for inferring precedence rules)

23.5.2 Usage

Information contained in an operator specification is used automatically in various ways during lexical analysis, parsing, and evaluation. The parse tree for an expression containing operators is a tree of token records as documented in Section 23.4.3, with a token record corresponding to each operator in the expression. These token records are derived from the operator specification with appropriate preprocessor and semantic fields as explained below.

Precedence

The last three fields in an operator record, loose, tight, and peer, affect the operator precedence, which affects the way parse trees are built. Any time one of these fields is changed as a result of the --operators command line option for any operator, the rules are updated automatically.

- Use of the peer field is the recommended way of establishing the precedence of a new operator rather than changing the precedence rules directly as in Section 23.2, because it is conducive to more consistent rules and is less likely to cause backward incompatibility.
- The loose field should have a true value only for declaration operators such as :: and =. However, some hand coded modifications to the compiler would also be required in order to introduce new kinds of declarations, making this field inappropriate for use in conjunction with the --operators command line option.
- The tight field is false for all operators except the very high precedence operators tilde (~), dash (-), library (..), and function application when expressed without a space, as in f(x). Otherwise, it is appropriate for infix operators whose left operand is rarely more than a single identifier.

Optimization

The list of functions in the optimizers field maps directly to the postprocessors field in a token record derived from an operator. An optimizer function can perform an arbitrary transformation on the result computed by the operator, but the convention is to restrict it to things that are in some sense "semantics preserving". In this way, the operator can be evaluated with or without the optimizer as appropriate for the situation.

Generally the operator semantics itself is designed as a function of manageable size in case it is to be stored or otherwise treated as data, while the optimizer associated with it may be a large or time consuming battery of general purpose semantics preserving transformations that are more convenient to keep separate. The latter is invoked only when the operator is associated with operands and evaluated at compile time. For most operators built into the default operator table, the result returned is a function, and the optimizer is the `optimization` function defined in the file `src/opt.fun`.

The reason for having a list of optimizers rather than just one is to cope with operators having a higher order functional semantics. For a solo operator ∇, the first optimizer in the list will apply to expressions of the form ∇x_0, the second to $(\nabla x_0)\ x_1$, and so on. In many cases, the `optimization` function is applicable to all orders.

Preprocessors

Because there is potentially a different semantics for each arity, the `preprocessor` in a `token` corresponding to an operator is automatically generated to detect the number and positions of the subtrees and to assign the `semantics` accordingly. Having done that, it will also apply the relevant function from the `preprocessors` field of the `operator` specification, if any.

The `preprocessors` in an operator specification are not required and should be used sparingly when defining new operators, because top-down transformations on the parse tree can potentially frustrate attempts to formulate a compositional semantics for the language, making it less amenable to formal verification. However, there are two reasons to use them somewhat more frequently.

One reason is to insert a so called "spacer" token into the parse tree using a function such as the following for a postfix preprocessor.

```
~lexeme=='(spacer)'?vhd/~& &vh:= ~&v; //~&V token[
   lexeme: '(spacer)',
   semantics: ~&h!]
```

The spacer should be inserted into the parse tree below any operator token that evaluates to a function but takes an operand that is not necessarily a function. such as the `!` and => operators. Normally if all nodes in a parse tree have the same postprocessors, they are deleted from all but the root to avoid redundant optimization. The spacer token performs no operation when the parse tree is evaluated other than to return the value of its subexpression, but its presence allows the subexpression to be optimized by its `optimizer` functions if applicable because they will not be deleted when the spacer is present.

The other reason to use preprocessors in an operator specification is in certain aggregate operators that reduce to the identity function if there is just one operand, such as cumulative conjunction, which can benefit from a preprocessor like this.

```
||~& -&~&d.lag-suffix.&Z,~&v,~&vtZ,~&vh&-
```

Algebraic properties

The dyadic field stores the information in Table 5.7 for each operator. For example, if an operator with a specification o is postfix dyadic, then ~dyadic.postfix o will be true. This information is not mandatory when defining an operator but may improve the quality of the generated code if it is indicated where appropriate. The field is referenced by the preprocessor of the function application operator defined in the file apt.fun.

Options

The options field in an operator record is of the same type as the suffix field in a token derived from it, but the options fields contains the set of all possible suffix elements for the operator, and the suffix field contains only those appearing in the source text for a given usage.

The options are a list of the form <$s_0 : x_0 \dots s_n : x_n$>, where each s_i is a character string containing exactly one character, and the x_i values can be of any type. For example, some operators allowing pointer suffixes have the list of pnodes as their options (see Section 23.1), and other operators that allow type expressions as suffixes have the type_constructors as their options, the main table of type_constructor records defined in the file tco.fun. Still others such as the /* operator have a short list of functional options defined as follows,

```
<'*': *,'=': ~&L+,'$': fan>
```

and other operators such as |= have combinations of these. However, no options should be specified for aggregate operators (e.g., parentheses and brackets) because they have a consistent style of using periods for suffixes as documented in Section 5.2.3, which is handled automatically.

The use made of the options by the operator depends on their type and the operator semantics, as explained further below. For example, a list of pnodes can be assembled into a pointer or pseudo-pointer by the percolation function defined in the file psp.fun, and a list of type constructors is transformed to a type expression or type induced function by the execution function defined in tag.fun. A list of functional combinators such as those above might only need to be composed with the operator semantic function.

Whatever options an operator may have, they should be documented in a few lines of text stored in the opthelp field, so that users are not forced to read the source code or search for a reference manual that might not exist or be out of date. The contents of this field are displayed when the compiler is invoked with the command line option --help suffixes, with the text automatically wrapped to fit into eighty columns on a terminal.

Semantics

The functions in the meanings field follow a variety of calling conventions depending on the arity and depending on whether the options field is empty.

If the `options` field is empty, the infix semantic function (i.e., the value accessed by `~meanings.infix` o for an operator o) takes a pair (x, y) as an argument, the prefix and postfix functions take a single argument x, and the aggregate semantic function takes a list of values <$x_0 \dots x_n$>. The contents of `~meanings.solo` o is not a function but simply the value obtained for the operator when it is used without operands, if this usage is allowed.

If there are options, then these fields are treated as higher order functions by the compiler, or as a first order function in the case of the solo arity. The argument to each function is the list of options following it in the source text, which will be members of the `options` field of the form s_i: x_i. Given this argument, the function is expected to return a function following the calling convention described above for the case without options.

As a short example, the infix semantic function for the assignment operator (`:=`) has the following form, and something similar is done for any operator allowing a pointer expression as a postprocessor.

```
~&lNlXBrY+percolation+~&mS; ~&?=/assign! "d". "d"++ assign
```

The `percolation` function takes a list of `pnode` records, which in this case will come from the suffix applied to the `:=` operator where it is used in a source text. It returns a pair (p, f) with a pointer p or a function f, at most one non-empty, depending on whether a pointer or a pseudo-pointer is detected. The `~&lNlBrY` function forms either the deconstructor function $\tilde{}p$ or takes the whole function f as the case may be. If this turns out to be the identity function, no postprocessing is required, so the semantics reduces to the virtual machine's `assign` combinator. Otherwise, the semantics takes a pair (x, y) to a function d+ `assign(`x`,`y`)`, where d is the function derived from the suffix.

Lexical analysis

The `mnemonic` and `excluder` fields in an `operator` specification map directly to the `lexeme` and `exclusions` fields in the token derived from it.

Mnemonics A new operator mnemonic can break backward compatibility even if it is not previously used, by coinciding with a frequently occurring character combination. For example, `$[` would be a bad choice for an operator because this character combination occurs frequently in the expression of record valued functions. If this combination started to be lexed as an operator, many existing applications would need to be edited.

Exclusions The `excluder` field can be used in operators with suffixes to suppress interpretation of a suffix. This function is consulted by the lexical analyzer when the operator lexeme is detected, and passed the string of characters following the lexeme up to the end of the line. If the function returns a true value, then the operator is considered not to have a suffix. One example is the assignment operator, `:=`, whose excluder detects the condition `~&ihB-='0123456789'`. This condition allows expressions such as f`:=0!` to be interpreted in the more useful sense, rather than having `0` as a pointer suffix.

Listing 23.8 a user defined tree mapping operator

```
#import std
#import nat
#import psp
#import ogl

#binary+

tm =

~&iNC operator[
   mnemonic: '^-',
   peer: '*^',
   dyadic: mode[solo: &],
   options: pnodes,
   opthelp: <'a pointer expression serves as a postprocessor'>,
   help: mode[
      infix: 'f^-g maps f to internal nodes and g to leaves in a tree',
      prefix: '^-g maps g only to terminal nodes in a tree',
      postfix: 'f^- maps f only to non-terminal nodes in a tree',
      solo: '^- (f,g) maps f to internal nodes and g to leaves'],
   meanings: ~&H\-+~&lNlXBrY,percolation,~&mS+- mode$[
      infix: //+ "h". "h"++ *^0+ ^V\~&v+ ~&v?+ ~&d;~~,
      prefix: //+ "h". "h"++ *^0+ ^V\~&v+ ~&v?/~&d+ ~&d;,
      postfix: //+ "h". "h"++ *^0+ ^V\~&v+ ~&v?\~&d+ ~&d;,
      solo: //+ "h". "h"++ *^0+ ^V\~&v+ ~&v?+ ~&d;~~]]
```

23.5.3 User defined operator example

The best designed operators are not necessarily the most complex, but the most easily learned and remembered. For a seasoned user, use of the operator becomes second nature, and for an inexperienced user, the time spent consulting the documentation is well compensated by the programming effort it saves. Most operators should be polymorphic, designed to support classes of types rather than specific types.

Specification

A first attempt at an operator aspiring to these attributes is shown in Listing 23.8. This operator operates on trees or dual type trees. It is analogous to the `map` combinator on lists, in that it determines a structure preserving transformation wherein a single function is applied to multiple nodes.

The operator, expressed by the symbol `^-`, is chosen to have the same precedence as the `*^` operator, and allows four arities. In the infix form it satisfies these recurrences,

$$\begin{aligned}(f\,\hat{}\text{-}g)\ d\hat{}\texttt{:}\ \texttt{<>} &= (g\ d)\hat{}\texttt{:}\ \texttt{<>}\\ (f\,\hat{}\text{-}g)\ d\hat{}\texttt{:}\ (h\texttt{:}t) &= (f\ d)\hat{}\texttt{:}\ (f\,\hat{}\text{-}g)\star\ (h\texttt{:}t)\end{aligned}$$

which is to say that the user may elect to apply a different function to the terminal nodes than to the non-terminal nodes. Its other arities have these algebraic properties,

$$\begin{aligned} \texttt{\^{}-}g &\equiv (\texttt{\~{}\&})\texttt{\^{}-}g \\ f\texttt{\^{}-} &\equiv f\texttt{\^{}-(\~{}\&)} \\ (\texttt{\^{}-})\,(f,g) &\equiv f\texttt{\^{}-}g \end{aligned}$$

the last being the solo dyadic property. Furthermore, the operator allows a pointer expression as a suffix, which can perform any postprocessing operations.

The question of whether these algebraic properties are most convenient would be resolved only by experience, so this specification allows design changes to be made easily and transparently. A postfix dyadic semantics, for example, would be achieved by substituting

```
"h". "f". "g". "h"+ *^0 ^V\~&v ~&v? ~&d;~~ ("f","g")
```

into the `meanings.postfix` function specification.

Demonstration

The code shown in Listing 23.8, stored in a file named `tm.fun`, is compiled as follows.

```
$ fun psp ogl tm.fun
fun: writing `tm'
```

To demonstrate the operator, we use a function `~&ixT^-`, in which the operand is a function that generates a palindrome by concatenating any list with its reversal. This expression is applied to a randomly generated tree of character strings.

```
$ fun --operators ./tm --m="~&ixT^- 500%sTi&" --c %sT
'zDOgcmHp}<eQQe<}pHmcgODz'^: <
   '-n.ss.n-'^: <
      '#A%WYSD-``-DSYW%A#'^: <'p'^: <>>,
      'PzT$&&$TzP'^: <
         'GV+qswwsq+VG'^: <
            ''^: <''^: <>,'Q'^: <>,''^: <>,''^: <>>,
            ^: (
               '}AL|yTm[[mTy|LA}',
               <'P'^: <>,~&V(),'P'^: <>,''^: <>>),
            ''^: <>>,
         'z/e4L'^: <>,
         'zg'^: <>>,
      'W'^: <>>,
   '22O'^: <>>
```

This result shows that all of the non-terminal nodes in the tree are palindromes.

23.6 Command line options

Most command line options to the compiler are not hard coded but based on executable specifications stored in a table.[1] The table can be dynamically modified by way of the `--formulators` command line option so as to define further command line options. In fact, all other command line options described in this chapter could be defined if they were not built in, and can be altered in any case.

23.6.1 Option specifications

Each command line option is specified by a record of type `_formulator` as defined in the file `src/for.fun`. This record contains the semantic function of the option, among other things, which works by transforming a record of type `_formulation` as defined in the file `mul.fun`. The latter contains dynamically created copies of all tables mentioned in previous sections of this chapter, as well as entries for user supplied functions that can be invoked during various phases of the compilation.

To be precise, the `formulator` record contains the following fields.

- `mnemonic` – a character string giving the full name of the option as it appears on the command line
- `filial` – a boolean value that is true if the option takes a file parameter
- `formula` – the semantic function of the option, taking an argument

  `((<`⟨*parameter*⟩ ...`>,`⟨*file*⟩`),`⟨*formulation*⟩`)`

  of type `((%sL,_file%Z)%X,_formulation)%X` and returning a new record of type `_formulation` derived from the argument
- `extras` – a list of strings giving the names of the allowable parameters for the option, currently used only for on-line documentation
- `requisites` a list of strings giving the names of the required parameters for the option, currently used only for on-line documentation
- `favorite` – a natural number specifying the precedence for disambiguation, with greater numbers implying higher precedence
- `help` – a character string containing a short description of the option for on-line documentation

The most important field of the `formulator` record is the `formula`, which alters the behavior of the compiler by effecting changes to the specifications it consults in the `formulation` record. Before passing on to a description of this data structure, we may note a few points about some of the remaining fields.

[1] The exceptions are the `--phase` option and to some extent the `--trace` option.

Command line parsing is handled automatically even in the case of user defined command line options. The `filial` field is an annotation to the effect that the command line is expected to contain the name of a file immediately following the option thus described. If such a file name is found, the file is opened and read in its entirety into a record of type `_file` as defined in the standard library. This record is then passed to the `formula`.

The parameters passed to the `formula` are similarly obtained from any comma separated list of strings following the option mnemonic on the command line, preceded optionally by an equals sign.

Recognizable truncations of the `mnemonic` field on the command line are acceptable usage, with no further effort in that regard required of the developer.

23.6.2 Global compiler specifications

The `formulation` data structure specifies a compiler by way of the following fields. Changing this data structure changes the behavior of the compiler.

- `command_name` – a character string containing the command whereby the compiler is invoked and diagnostics are reported
- `source_filter` – a function taking a list of input files (type `_file%L`) to a list of input files, invoked prior to the initial lexical analysis phase
- `token_filter` – a function taking the initial a list of lists of lists of tokens (type `_token%LLL`) to a result of the same type, invoked after lexical analysis but before parsing
- `preformer` – a function taking a list of parse trees before preprocessing to a list of parse trees
- `postformer` – a function taking a parse tree for the whole compilation after preprocessing stabilizes to a parse tree suitable for evaluation
- `target_filter` – a function taking a list of output files to a list of output files, invoked after all parsing and evaluation
- `import_filter` – a function for internal use by the compiler (refer to the source code documentation in `src/mul.fun`)
- `precedence` – a quadruple of pairs of lists of strings describing precedence rules as defined in Section 23.2.
- `operators` – the main list of operators, with type `_operator%L` as defined in Section 23.5.1.
- `directives` – the main list of compiler directives, type `_directive%L` as defined in Section 23.4.
- `formulators` – the list of compiler option specifications, `_formulator%L` as defined in Section 23.6.1.

- `help_topics` – a module of functions (type `%fOm`) each associated with a possible parameter to the `--help` command line option, as documented in Section 23.7.

Conspicuous by their absence are tables for the type constructors and pointer operators. These exist only in the `suffix` fields of individual operators in the table of operators. Extensions of the language involving new forms of operator suffix automata would require no modification to the main `formulation` structure (although a new help topic covering it might be appropriate, as explained in Section 23.7).

All of the functional fields in this structure are optional and can be left unspecified. The default values for most of them are the identity function. However, in order for command line options to work well together, those that modify the filter functions should compose something with them rather than just replacing them. For example, in an option that installs a new token filter, the `formula` field should be a function of the form

```
&r.token_filter:=r +^\-|~&r.token_filter,! ~&|- ~&l; ...
```

where the remainder of the expression takes a pair (p, f) of a list of parameters p and possibly a configuration file f to a function that is applied to the token stream.

Token streams

The token stream is represented as a list of type `_token%LLL` because there is one list for each source file. Each list pertaining to a source file is a list of lists of tokens. Each list within one of these lists represents a contiguous sequence of tokens without intervening white space. Where white space or comments appear in the source file, the token preceding it is at the end of one list and the token following it is at the beginning of the next. Hence, a source code fragment like `(f1, g2)`, would have the first four tokens together in a list, and the next three in the subsequent list.

Parse trees

Parse trees follow certain conventions to express distinctions between operator arities, which must be understood to manipulate them correctly. If a user supplied function is installed as the `preformer` in the `formulation` record, its argument will be a list of parse trees as they are constructed prior to any self-modifying transformations determined by the `preprocessor` field in the `token` records. Prior to preprocessing, every operator token initially has two subtrees.

- For infix operators, the left operand is first in the list of subtrees and the right operand is second.
- For prefix operators, the first subtree is empty and the second subtree is that of the operand.
- For postfix operators, the first subtree contains the operand and the second subtree is empty.

Listing 23.9 parse tree for a prefix operator `%=s`, showing an empty first subexpression

```
ˆ: (
   token[
      lexeme: '%=',
      location: (2,7),
      preprocessor: 983811%fOi&],
   <
      ˜&V(),
      ˆ:<> token[
         lexeme: 's',
         location: (2,9)]>)
```

Listing 23.10 parse tree for a postfix operator `s%=`, showing an empty second subexpression

```
ˆ: (
   token[
      lexeme: '%=',
      location: (2,8),
      preprocessor: 983811%fOi&],
   <
      ˆ:<> token[
         lexeme: 's',
         location: (2,7)],
      ˜&V()>)
```

Listing 23.11 parse tree for an infix operator `s%=t`, with two non-empty subexpressions

```
ˆ: (
   token[
      lexeme: '%=',
      filename: 'command-line',
      location: (2,8),
      preprocessor: 983811%fOi&],
   <
      ˆ:<> token[
         lexeme: 's',
         location: (2,7)],
      ˆ:<> token[
         lexeme: 't',
         location: (2,10)]>)
```

These conventions are illustrated by the parse trees shown in Listings 23.9, 23.10, and 23.11. The operator `%=` has the same lexeme in all three arities, but the infix, prefix, or postfix usage is indicated by the subtrees.

For aggregate operators such as parentheses and braces, the enclosed comma separated sequence of expressions is represented prior to preprocessing as a single expression in which the comma is treated as a right associative infix operator. The left enclosing aggregate operator is parsed as a prefix operator and stored at the root of the tree. The matching right operator is parsed as a postfix operator and stored at the root of the second subtree. Compiler directives such as `#export+` and `#export-` are parsed the same way as aggregate operators. An example of a parse tree in this form is shown in Listing 23.12.

It can also be seen from these examples that most operator tokens initially have a `preprocessor` but no `semantics`. The semantics depends on the operator arity, which is detected by the `preprocessor` when it is evaluated. At a minimum, the preprocessor for each operator token initializes its `semantics` field for the appropriate arity, deletes any empty subtrees, and usually deletes the preprocessor itself as well. The preprocessor for an aggregate operator will check for a matching operator and delete it if found. It will also remove the comma tokens and transform their subexpressions to a flat list.

It is important to keep these ideas in mind if a user supplied function is to be installed as the `postformer` field, whose argument will be a parse tree in the form obtained after preprocessing. An example is shown in Listing 23.13.

23.6.3 User defined command line option example

We conclude the discussion of command line options with the brief example of a user defined command line option shown in Listing 23.14. The code shown in the listing provides the compiler with a new option, `--log`, which causes an extra annotation to be written to the preamble of every generated binary or executable file stating the names of all source files given on the command line. This information could be useful for a "make" utility to construct the dependence graph of modules in a large project.

Theory of operation

There could be several ways of accomplishing this effect, but the basic approach in this case is to alter the `postformer` field of the compiler's specification. The function in this field takes the main parse tree after preprocessing but before evaluation. At this stage the parse tree will consist only of directives and declarations (i.e., = operator tokens) whose subexpressions have been reduced to single leaf nodes by evaluation.

The first step is to form the set of file names by collecting the `filename` fields from all tokens in the parse tree, formatted into a string prefaced by the word "`dependences:`". Next, the function is constructed that will insert this string into the preamble of each file in a list of files. Executable files require slightly different treatment than other binary files, because the last line of the preamble in an executable file must contain the shell command to launch the virtual machine, so the annotation is inserted prior to the last line.

Listing 23.12 the parse tree for {a,b,c}, showing commas and aggregate operators

```
^: (
   token[
      lexeme: '{',
      location: (2,7),
      preprocessor: 154623%fOi&],
   <
      ~&V(),
      ^: (
         token[
            lexeme: '}',
            location: (2,13),
            preprocessor: 152%fOi&,
            semantics: 5%fOi&],
         <
            ^: (
               token[
                  lexeme: ',',
                  location: (2,9),
                  semantics: 177%fOi&],
               <
                  ^:<> token[
                     lexeme: 'a',
                     location: (2,8)],
                  ^: (
                     token[
                        lexeme: ',',
                        location: (2,11),
                        semantics: 177%fOi&],
                     <
                        ^:<> token[
                           lexeme: 'b',
                           location: (2,10)],
                        ^:<> token[
                           lexeme: 'c',
                           location: (2,12)]>)>),
            ~&V()>)>)
```

Listing 23.13 the parse tree from Listing 23.12 after preprocessing

```
^: (
   token[
      lexeme: '{',
      location: (2,7),
      preprocessor: 852%fOi&,
      postprocessors: <0%fOi&>,
      semantics: 480%fOi&],
   <
      ^:<> token[
         lexeme: 'a',
         location: (2,8)],
      ^:<> token[
         lexeme: 'b',
         location: (2,10)],
      ^:<> token[
         lexeme: 'c',
         location: (2,12)]>)
```

Listing 23.14 command line option to add source dependence information to output files

```
#import std
#import lag
#import for
#import mul

#binary+

log =

~&iNC formulator[
   mnemonic: 'log',
   formula: &r.postformer:=r +^\-|~&r.postformer,! ~&|- ! -+
      ~&ar^& ~lexeme.&ihB==`#?ard(
         &ard.postprocessors:=ar ~&iNC+ ^|/~&+ ~&al,
         ~&ard2falrvPDPMV),
      _token%TfOwXMk+ ^\~& -+
         ~&iNC; "d". * ~preamble?\~& preamble:= ~preamble; ?(
            -&~&h=]'!/bin/sh',~&z=]'exec avram',~&yzx=]'\'&-,
            ^T/~&yyNNCT ((* :/` ) "d")--+ ~&yzPzNCC,
            --<''>+ --((* :/` ) "d")+ ~&iNNCT),
         'dependences: '--+ mat` + ~&s+ *^0 :^\~&vL ~&d.filename+-+-,
   help: 'list source file dependences in executables and libraries']
```

The `postformer` will descend the parse tree from the root, stopping at the first directive token, and reassign its `postprocessors` to incorporate the preamble modifying function just constructed. An alternative would have been to change the `semantics` function, but this approach is more straightforward.

By convention, every parse tree whose root is a directive token (i.e., a token whose lexeme begins with a hash and is derived from a compiler directive in the source code) evaluates to a pair (s, f), where s is a list of assignments of identifiers to values (type `%om`), and f is a list of files (type `_file%L`). The assignments in s are obtained from the declarations within the scope of the directive, and the files in f are those generated by the directive at the root or by other output file generating directives in its scope. It therefore suffices for the head postprocessor to be a function of the form `^|/~&` d, so as to pass the left side of its argument through to its result, and to apply the preamble modifying function d to the right.

Demonstration

The binary file containing the new command line option is easily prepared as shown.

```
$ fun lag for mul log.fun
fun: writing `log'
```

One might then test it on itself.

```
$ fun --formulators ./log lag for mul log.fun --log
fun: writing `log'
$ cat log
#
#
# dependences:  for lag log.fun mul nat std
#
syCs{auXn[eWGCvbVB@wDt...
```

23.7 Help topics

The `--help-topics` command line option requires a binary file as a paramter containing a list of assignments of strings to functions (type `%fm`). For each item s: f of the list, the function f takes an argument of the form

$$\texttt{(<}\langle \textit{parameter} \rangle \ldots \texttt{>,} \langle \textit{formulation} \rangle \texttt{)}$$

to a list of character strings to be displayed when the compiler is invoked with the option `--help` s. That is, the string s is a possible parameter to the `--help` command line option. The parameters in the argument to f are any further parameters that may appear after s in a comma separated sequence on the command line.

The default help topics are automatically updated when any change is made to the operators, directives, or formulators (and by extension, to the types or pointer constructors), as

Listing 23.15 a user defined help topic

```
#import std
#import nat
#import for
#import mul

#binary+

pri =

~&iNC 'priority': ~&r.formulators; -+
   ^plrTS(
      (--'  '+ ~&rS+zipp` )^*D(leql$^,~&)+ <'option','------'>--+ ~&lS,
      <'priority','--------'>--+ ~&rS; * ~&h+ %nP),
   ~&rF+ * ^/~mnemonic ~favorite+-
```

shown in previous examples. This option is needed therefore only if a whole new classification of interactive help is intended, such as might arise if the language were extensively customized in other respects.

Listing 23.15 shows a small example of how a user defined help topic can be specified. Recall that certain command line options have a higher disambiguation priority than others (page 277), but that this information is accessible only by consulting the written documentation, which may be unavailable or obsolete. To correct this situation, the help topic defined in Listing 23.15 equips the compiler with an option `--help priority`, which will display the priorities of any command line options with priorities greater than zero.

The operation of the code is very simple. It accesses the `formulators` field in the main `formulation` record that will be passed to it as its right argument, filters those with positive `favorite` fields, and displays a table showing the mnemonics and the priorities of the results. This code can be tested as follows.

```
$ fun for mul pri.fun
fun: writing `pri'
$ fun --help-topics ./pri --help priority

option     priority
------     --------
help       1
parse      1
decompile  1
archive    1
optimize   1
show       1
cast       1
```

Where are you going with this, Ikea boy?

Brad Pitt in *Fight Club*

24 Manifest

This chapter gives a general overview of the compiler source organization for the benefit of developers wishing to take it further. The compiler consists of a terse 6305 lines of source code at last count, written entirely in Ursala, divided among 25 library files and a very short main driver shipped under the `src` directory of the distribution tarball. These statistics do not include the standard libraries documented in Part III, except for `std.fun` and `nat.fun`.

Library files are employed as a matter of programming style, not because the project is conceived as a compiler developer's tool kit. Most library functions are geared to specific tasks without much scope for alternative applications. Nor is there any carefully planned set of abstractions meant to be sustained behind a stable API. Nevertheless, this material may be of interest either to developers inclined to make small enhancements to the language not covered by features discussed in the previous chapter, or to those concerned with scavenging parts of the code base for a new project.

Comprehensive developer level documentation of the compiler will probably never exist, because it would double the length of this manual, and because not much of the code is amenable to natural language descriptions in any case. Moreover, many parts of the compiler perform quite ordinary tasks that a competent developer could implement in various ways more easily than consulting a reference. Furthermore, to the extent that any such documentation is useful, it necessarily renders itself obsolete. We therefore limit the scope of this chapter to a brief summary of each library module in relation to the others.

Table 24.1 lists the compiler modules in the `src` directory with brief explanations of their purposes. Generally modules in the table depend only on modules appearing above them in the table, although there are cyclic dependences between `std` and `nat`, between `tag` and `tco`, and between `for` and `mul`.

The intermodular dependences are documented in the executable shell script named `bootstrap`, also distributed under the `src` directory. Execution of this script will re-

module	comment
`cor`	virtual machine combinator mnemonics
`std`	standard library
`nat`	natural number library
`com`	virtual machine combinator emulation
`ext`	data compression functions
`pag`	parser generator
`opt`	code optimization functions
`sol`	fixed point combinators
`tag`	type expression supporting functions
`tco`	table of type constructors
`psp`	table of pointer operators
`lag`	lexical analyzer generator
`ogl`	operator infrastructure
`ops`	main table of operators
`lam`	parse tree transformers for lambda abstraction
`apt`	specifications of invisible operators
`eto`	specification of declaration operators
`xfm`	symbol name resolution and substitution functions
`dir`	table of compiler directives
`fen`	parser and lexical analysis drivers and glue code
`pru`	precedence rule specifications
`for`	supporting functions for command line options
`mul`	compiler formulation data structure declaration
`def`	main table of command line options
`con`	command line parsing and glue code
`fun`	executable driver

Table 24.1: compiler modules

build the compiler from source, but depends on the `fun` executable. The script has a command line option to generate a compiler with extra profiling features, also documented within.

A full build is an over night job, subject to performance variations, of course. Most of the CPU time for a build is spent on code optimization, and the next largest fraction on file compression. Any production version of the compiler will bootstrap an exact copy of itself, unless the time stamp on `for.fun` has changed. Some modifications to the source code may require multiple iterations of bootstrapping in order for the compiler to recover itself.

The `cor`, `std`, and `nat` modules are previously documented in Listing 3.1 and Chapters 8 and 9. The remainder of this chapter expands on Table 24.1 with some more detailed comments on the other modules.

24.1 `com`

One way to simplify the job of implementing an emulator for the virtual machine is to code the smallest subset of combinators necessary for universality, and arrange for the remainder to be translated dynamically into these. The `com` module contains a selection of virtual machine code transformaters relevant to this task. For example, a program of the form `iterate(`p`,`f`)` using the virtual machine's `iterate` combinator can be transformed into one using only recursion.

The `rewrite` function automatically detects the root combinator of a given program and transforms it if possible. This function is written to an external file as a C language character constant when this library is compiled, which is used by `avram` as a sort of virtual "firmware" in the main evaluation loop.

The other use of this module is in the `opt` code optimization module (Section 24.4), where it is used for abstract interpretation when optimizing higher order functions.

24.2 `ext`

This module contains the data compression functions used with compressed types (t`%Q`), archived libraries, and self-extracting executables. Compression is a bottleneck in large compilations that would reward a faster implementation of these functions with noticably better performance.

The compression algorithm transforms a given tree t to a tuple $((p, s), t')$ if doing so will result in a smaller size, or to $((), t)$ otherwise. The tree t' is like t with all occurrences of its maximum shared subtree deleted. The subtree s is that which is deleted, and p is another tree identifying the paths from the root to the deleted subtrees in t', similarly to a pointer constant. The tuple $((p, s), t')$ itself usually can be compressed further in the same way, so the algorithm iterates until a fixed point is reached or until the size of the largest shared subtree falls below a user defined threshold.

Most of the time in this algorithm is spent searching for the maximum shared subtree. A data structure consisting of eight queues is used for performance reasons, although any positive number would also work. Each queue contains a list of lists of subtrees. Each subtree has the same weight as the others in its list, and the lists are queued in order of decreasing member tree weights. The residual of each tree weight modulo 8 is the same as that of all other trees within the same queue.

The algorithm begins with all but one queue empty, and the non-empty one containing only a single list containing a single tree, which is the tree whose maximum shared subtree is sought.

On each iteration, the list containing the heaviest trees is dequeued, and inspected for duplicates. If a duplicated entry is found, it is the answer and the algorithm terminates. Otherwise, every tree in the list is split into its left and right subtrees, these are inserted in their appropriate places in the existing data structure, and the algorithm continues.

The paths p for the shared subtree obtained above are not recorded during the search, but detected by another search after the subtree is found.

This algorithm relies heavily on the fact that computing tree weights and comparison of trees are highly optimized operations on the virtual machine level. It is faster to recompute the weight of a given tree using the `weight` combinator than to store it.

24.3 `pag`

This module contains a generic parser generator based on an *ad hoc* theory, taking a data structure of type `_syntax` describing the grammar of the language as input. Traditional parser generator tools are inadequate for the idiosyncrasies of Ursala with regard to operator arity and overloading, but a hand coded parser would be too difficult to maintain, especially with user defined operators.

The parsers generated by this method are much like traditional bottom-up operator precedence parsers using a stack, but are generalized to accommodate operator arity disambiguation on the fly and a choice of precedence relations depending on the arities of both operators being compared.

Rather than taking a list of tokens as input, the parser takes a list of lists of tokens, with white space implied between the lists, but juxtaposition of the tokens within each list (see page 457). Each token is first annotated with a list of four boolean values to indicate its possible arities prior to disambiguation. This information is derived partly from the operator specifications encoded by the `syntax` record parameterizing the parser, and partly by contextual information (for example, that the last token in a list can't be a prefix operator unless it has no other arity). A token is ready to be shifted or reduced only when all but one of its flags are cleared. Otherwise a third alternative, namely a disambiguation step, is performed to eliminated at least one flag by contextual information that may at this stage depend on the stack contents.

An exception to the conventional operator precedence parsing rules is made when a prefix operator is followed by a postfix operator and both are mutually related in precedence. In this case, they are simulataneously reduced, so that expressions like `<>` or `{}` can be parsed as required. This test also applies to prefix and postfix operators with an expression between them, wherein the reduction results in a parse tree like that of Listing 23.12.

Although the `syntax` data structure doesn't explicitly represent any distinction between aggregate operators and ordinary prefix or postfix operators, aggregate operators are indicated by being mutually related with respect to prefix-postfix precedence. There is never a need for this condition to hold with other prefix or postfix operators, because the relation is meaningful only in one direction.

24.4 `opt`

Code optimization functions are stored in the `opt` library module. The optimizations are concerned with transforming virtual machine code to simpler or more efficient forms while preserving semantic equivalence.

Optimizations include things like constant folding, boolean and first order logic simplifications, factoring of common subexpressions, some forms of dead code removal, and other *ad hoc* transformations pertaining to list combinators and recursion. The results are not provably optimal, which would be an undecidable problem, but are believed to be semantically correct and generally useful. A more rigorous investigation of code optimization for this virtual machine model awaits the attention of a suitably qualified algebraist.

An intermediate representation of the virtual machine code is used during optimization, which is a tree of combinators (type `%sfOZXT`) as explained on pages 86 and 141. The left of each node is a mnemonic from the `cor` library, and the right is a function that will transform this representation to virtual code given the virtual code for each subtree.

There are further possibilities for optimization of higher order functions. A second order function in this tree representation can be evaluated with a symbolic argument by abstract interpretation. Several functions concerned with abstract interpretation are defined in the library. The result, if it is computable, will be the representation of a first order function in which some of the nodes contain an unspecifed semantic function. Optimization in this form followed by conversion back to second order often will be very effective.

This technique generalizes to higher orders, but the drawback is that it is not possible to infer the order of a function by its virtual code alone, and mistakenly assuming a higher order than intended will generally incur a loss of semantic equivalence. In certain cases the order can be detected from source level clues, such as functions defined by lambda abstraction or functions using operators implying a higher order. The `#order+` compiler directive, which is currently unused, could serve as a pragma for the programmer to pass this information to the optimizer.

Code optimization is an interesting area for further work on the compiler, but should not be pursued indiscriminately. Optimizations that are unlikely to be needed in practice will serve only to slow down the compiler. Introduction of new optimizations that conflict with existing ones (i.e., by implying incompatible notions as to what constitutes optimality) can cause non-termination of the optimizer. Of course, semantically incorrect "optimizations" can have disastrous consequences. Any changes to the optimization routines should be validated at a minimum by establishing that the compiler exactly reproduces itself with sufficiently many iterations of bootstrapping.

24.5 sol

The main purpose of this library module is to implement the algorithm for general solution of systems of recurrences. The #fix compiler directive documented in Section 7.5.3 is one source level interface to this facility, and the use of mutually dependent record declarations is the other (page 158). The general_solution function takes a list of equations and user defined fixed point combinators to its solution following a calling convention with detailed documentation in the source, including a worked example.

The general solution algorithm consists mainly of term rewriting iterations necessary to separate a system of mutually dependent equations to equations in one variable. Following that, obtaining the solutions is a straightforward application of each equation's respective fixed point combinator. Thorough exposition of the algorithm is a subject for a separate article. However, being only sixteen lines of code and embedding many typed breakpoints of the style described starting on page 145, its inner workings are easily open to inspection.

This module also includes the function_fixer and fix_lifter functions explained in Section 7.5.3.

24.6 tag

This module contains some functions relevant to type expressions, and also contains the declaration of the type_constructor record.

Many of the functions defined in this module underlie the instance generators of primitive types and type constructors, along with their statistical distributions. These properties are adjustable only by hard coded changes to the compiler source through this module.

Miscellaneous functions used in the definitions of various type constructors are also present, as is the execution function, which builds a type expression from a list of constructors by executing their microcode (see page 436). This function is needed to define the semantics of operators allowing type expressions as suffixes (e.g., the % and %- operators, Section 6.11.2).

The fixed point combinators general_type_fixer and lifted_type_fixer are also defined in this module. These are used internally by the compiler for solving systems of mutually dependent record declarations, but may also be of some use to developers wishing to construct mutually recursive types explicitly.

24.7 tco

This library module contains the main table of type constructors. Adding a user defined type constructor to this table and rebuilding the compiler can be done as an alternative to loading one dynamically from binary a file as described in Section 23.3. The effect will be that the user defined type constructor becomes a permanent feature of the language.

24.8 psp

This module contains the main table of pointer constructors, the declaration of the pnode record type specifying pointer constructors, and the percolation function used to translate a list of pointer constructors to its pointer or pseudo-pointer functional semantics. The percolation function is used in the definition of any operator that allows a pointer expression as a suffix.

Adding a user defined pointer constructor to this table can be done as an alternative to loading it from a binary file as described in Section 23.1. The effect will be to make it a permanent feature of the language. As discussed previously, there are no unused pointer mnemonics remaining, and changing an existing one will break backward compatibility. However, an unlimited number of escape codes can be added, which would be done by appending more pnode records to the escapes table in the source.

24.9 lag

Functions pertaining to lexical analysis are stored in the lag library. This library also includes the declaration of the token record type, and a few operations on parse trees.

Lexical analysis is less automted than parsing (Section 24.3), requiring essentially a hand coded scanner for each lexical class (e.g., numbers, strings, *etcetera*) although some of these functions are parameterized by lists of operators or directives derived automatically from tables defined elsewhere.

The scanner for each lexical class consists of a triple (n, p, f) called a "plugin", where n is a natural number describing the priority of the scanner, p is a predicate to detect the class, and f is a function to lex it. The functions p and f take an argument of type %nWsLLXJ of the form ~&J(h, (l,c),<s...>), where refer(h) is the lexical analyzer meant to be called recursively, l and c are the line and column numbers of the current character in the input stream, and s is the current line of the input stream beginning with the current character.

The function p is supposed to return a boolean value that is true if s begins with an instance of the lexical class in question, and false otherwise.

The function f is applied only when p is true, and should return list of token records beginning with the one corresponding to the current position in the input stream, and followed by those obtained from a recursive call to h. That implies that a new argument of the form ~&J(h, (l',c'),<s'...>) must be constructed and passed in a recursive invocation of h, (usually of the form ^R/~&f...) with the line and column numbers adjusted accordingly, and the input stream advanced to the character past the end of the current token. Alternatively, if an error is detected, f can raise an exception, but should include the successors of the line and column numbers as part of the message.

Two other important functions in this library are `preprocess` and `evaluation`. The `preprocess` function takes a parse tree of type `_token%T` and transforms it under the direction of its internal preprocessor functions, as explained in Section 23.4.3. The `evaluation` function takes a parse tree to its value as defined by its `semantics` fields.

24.10 `ogl`

This library module contains the `operator` record type declaration (Section 23.5.1) and various functions in support of operator definitions.

One useful entry point is the `token_forms` function, which takes a list of operator records to a list of token records suitable for parameterizing the `built_ins` plugin of the `lag` module described in the previous section. Another is the `propagation` function, for operators allowing pseudo-pointers as operands, whose usage is best understood by looking at a few examples in the `ops` module.

24.11 `ops`

This module contains the main table of operators. Adding a new operator to this table and rebuilding the compiler is a more persistent alternative to loading a user defined operator from a binary file as described in Section 23.5.

Note that unlike operator specifications loaded from a file, these tables are fed through a function in the `default_operators` declaration that initializes the `optimizers` fields to copies of the `optimization` function defined in the `opt` module if they are non-empty. This feature is not necessarily appropriate if new operators are to be defined over non-functional semantic domains, and would require some minor reorganization.

24.12 `lam`

This module contains the code that allows functions to be specified by lambda abstraction. Lambda abstraction is a top-down source transformation implemented by a fairly simple algorithm. An expression of the form `("x","y"). f(g "x","y")`, for example, is transformed to `f^(g+ ~&l,~&r)`, with deconstructors replacing the variables, composition replacing application, and the couple operator used in application of functions of pairs. Subexpressions without bound variables are mapped to constant functions by the algorithm. The algorithm requires no modification if new operators are defined in the language, because their semantic functions are obtained from the `semantics` fields in the parse tree regardless.

Being a source transformation, the lambda abstraction code forms part of the preprocessor for the `.` operator, but because this operator is overloaded, the preprocessor is not defined until the arity is determined to be either postfix or infix. The postfix usage is initially parsed as a function application (e.g., `("x".)` *e*) with the implied application token at the root of the parse tree, so it becomes the responsibility the application token's preprocessor to reorganize the tree appropriately.

The virtual code generated by a naive implementation of the above algorithm tends to be suboptimal, so this library also includes several postprocessing transformations designed to improve the quality. These are semantically correct but do not always improve the code, and therefore can be disabled by the `#pessimize` directive.

24.13 apt

This module contains specifications for the tokens representing white space in a source file. There are three kinds of white space, which are the space between consecutive declarations, the space betwen a functional expression and its argument, and the space where there is insufficient information to distinguish between the two other cases. These are designated as `separation`, `application`, and `juxtaposition` respectively.

Only `application` has a meaningful semantics, while the other two are expected to be transformed out in the course of preprocessing and will raise an exception if they are ever evaluated.

The preprocessor of the `application` token is responsible for performing all algebraic transformations associated with dyadic operators. For this reason, the token is defined by way of a function that takes the main operator table as input, including any run time additions.

Several minor source level optimizations are also performed by the preprocessor of the `application` token, such as recognition of lambda abstraction as mentioned in the previous section, and elimination of binary to unary combinators in some cases. These transformations depend on some of the operators having the mnemonics they have, independently of the table of operators.

24.14 eto

This module defines the tokens associated with the declaration operators, = and `::`. These operators do not appear in the main table of operators but are defined instead in this module, mainly because their definitions are parameterized by the rest of the operators for various reasons.

The `::` operator has no semantics at all but only a preprocessor that transforms itself to a sequence of ordinary declarations in terms of the = operator, and also inserts `#fix` directives with appropriate fixed point combinators for types and functions in the event of self-referential declarations. It includes features to detect when a lifted fixed point combinator can be used in preference to an ordinary one to achieve the equivalent order, and uses it if possible (see Section 7.5.3 for theoretical background).

The = operator semantics follows a required convention of evaluating an expression to an assignment $s\colon x$, with s being the identifier and x being the value of the body of the expression. The preprocessor of this operator is complicated by the need to interact correctly with the `#pessimize` directive, and by the need to transform declarations like `f("x") = y` in conventional mathematical notation to the lambda abstraction `f = "x". y`.

Although this library is short, the code in it is more difficult than most and will yield only to a meticulous reading.

24.15 `xfm`

This library is concerned primarily with establishing the rules of scope described in Section 7.2 and with resolution of symbolic names as needed for evaluation of expressions. There are also functions concerned with dead code removal, and with invoking the general solution algorithm defined in the `sol` module (Section 24.5) when cyclic dependences are detected. The latter are applied globally to the parse tree of a given compilation in the `con` module (Section 24.22), whereas the former constitute the bulk of the preprocessor for the `#hide` directive defined in the `dir` library (Section 24.16).

24.16 `dir`

The `directive` record declaration describing compiler directives is declared in this module, as is the main table of compiler directives. Adding a user defined compiler directive specification to this table and rebuilding the compiler has a similar effect to loading a directive specification from a binary file as described in Section 23.4, except that in this case the directive will become a permanent feature of the language.

This library also declares a function called `token_forms`. Similarly to a function of the same name in `ogl` (Section 24.10), this function transforms a list of directive specifications to a list of tokens. The main purpose of this function is to construct the list of tokens used to parameterize the `directives` plugin in the lexical analyizer generator (Section 24.9), but it also has applications in various other contexts where there is a need to construct a parse tree containing directives.

24.17 `fen`

This module instantiates the parser and lexical analyzer generators of the `pag` and `lag` modules with the operators, directives, and precedence rules from `ops`, `eto`, `apt`, `dir`, and `pru`.

Certain other details are also addressed in this module, such as the precedence rules for such non-operators as white space, commas, smart comments (page 246), and dash bracket delimiters (page 118). The lexical analyzer produced by the `lexer` function in this module includes a hand written scanner that inserts `separation` tokens between consecutive declarations so that the automatically generated parser can apply to a whole file. The relaxation of the requirement that all compiler directives appear in matched opening and closing pairs is also a feature of this lexical analyzer, which inserts matching directives using a hand written algorithm.

24.18 pru

This module contains the main tables of precedence rules depicted in Tables 5.3 through 5.6, and also contains a function for pretty printing a parse tree, which is used by the `--parse` command line option. A function to compute the operator precedence equivalence classes shown in Table 5.2 is also included, but the underlying equivalence relation is determined by the `peer` fields of the operators defined in the `ops` module.

Redefining the operator precedence rules in this module followed by rebuilding the compiler can be done as an alternative to temporarily loading the rules from a file as explained in Section 23.2. The effect will be a permanent change in the operator precedence rules of the language. As noted previously, changes in precedence rules are likely to break backward compatibility.

24.19 for

This module contains the declaration of the `formulator` record used to describe command line options as explained in Section 23.6.1, and a couple of functions that are helpful for constructing records of this type. There are also some important constants declared in this module, such as the email address of the Ursala project maintainer, and the main compiler version number, which is displayed when the compiler is invoked with the `--version` option. The version number may also be supplemented with a time stamp, which is derived from the time stamp of this source file.

One function in this module, `directive_based_formulators`, takes a list of compiler directive specifications as input, and returns a list of `formulator` records. This function is the means whereby any compiler directive automatically induces a corresponding command line option.

Another function, `help_formulator`, takes a table of help topics as described in Section 23.7 and returns the formulator for the `--help` command line option parameterized by those topics.

24.20 mul

This very short module contains the declaration for the `formulator` record, which embodies a complete specification for the compiler by including all tables previously mentioned, as explained in Section 23.6.2. A couple of functions define default values for some of the formulation fields, and the `default_formulation` function takes a table of `formulator` records to a `formulation` using them.

24.21 `def`

The main tables of `formulator` records and help topics are stored in this module. These tables can be modified and the compiler rebuilt as an alternative to loading help topics or command line option specifications from a binary file as explained in Sections 23.6 and 23.7. In this case, the modifications will become permanent features of the compiler.

24.22 `con`

This module contains functions responsible for managing the main flow of control during a compilation. The `customized` function performs the initial interpretation of command line options and parameters to arrive at the `formulation` record that will be used subsequently.

Thereafter, compilation is divided into three main phases, corresponding to the results that can be inspected by the `--phase` command line option. The first covers lexical analysis and parsing. The second covers preprocessing, dependence analysis, and some local evaluation of expressions. The third phase includes all remaining evaluation and execution of compiler directives, and the construction of the list of output files.

Each of these phases is specified by one of the functions in the list of `phases`. These are higher order functions parameterized by a `formulation` record, which return functions operating on parse trees and files. The composition of these functions, achieved by the `compiler` function, constitutes the bulk of the compiler.

24.23 `fun`

This file contains the executable driver for the functions defined in the `con` module. The additional features implemented in this file are detection and handling of the `--phase` command line option, displaying the default help messages when no files or options are given, supporting the `command-name` feature of the `formulation` by incorporating it into diagnostic messages, displaying a warning when output generating directives are omitted, and trapping non-printing characters in diagnostic messages.

While it remains a burden assiduously avoided, it is not unexpected and thus not beyond a measure of control.

The Architect in *The Matrix Reloaded*

A
Changes

A problem with software documentation perhaps first observed by Gerald Weinberg is that if it's too polished, it gets out of sync with the software because it becomes intimidating for some people to update it.

This appendix is reserved for contributions by maintainers, site administrators, or anyone redistributing the software who is disinclined to alter the main text. Any commentary, errata, or documentation of new features recorded here should be deemed to take precedence.

B

GNU Free Documentation License

Preamble

The purpose of this License is to make a manual, textbook, or other functional and useful document “free” in the sense of freedom: to assure everyone the effective freedom to copy and redistribute it, with or without modifying it, either commercially or noncommercially. Secondarily, this License preserves for the author and publisher a way to get credit for their work, while not being considered responsible for modifications made by others.

This License is a kind of “copyleft”, which means that derivative works of the document must themselves be free in the same sense. It complements the GNU General Public License, which is a copyleft license designed for free software.

We have designed this License in order to use it for manuals for free software, because free software needs free documentation: a free program should come with manuals providing the same freedoms that the software does. But this License is not limited to software manuals; it can be used for any textual work, regardless of subject matter or whether it is published as a printed book. We recommend this License principally for works whose purpose is instruction or reference.

1. APPLICABILITY AND DEFINITIONS

This License applies to any manual or other work, in any medium, that contains a notice placed by the copyright holder saying it can be distributed under the terms of this License. Such a notice grants a world-wide, royalty-free license, unlimited in duration, to use that work under the conditions stated herein. The "**Document**", below, refers to any such manual or work. Any member of the public is a licensee, and is addressed as "**you**". You accept the license if you copy, modify or distribute the work in a way requiring permission under copyright law.

A "**Modified Version**" of the Document means any work containing the Document or a portion of it, either copied verbatim, or with modifications and/or translated into another language.

A "**Secondary Section**" is a named appendix or a front-matter section of the Document that deals exclusively with the relationship of the publishers or authors of the Document to the Document's overall subject (or to related matters) and contains nothing that could fall directly within that overall subject. (Thus, if the Document is in part a textbook of mathematics, a Secondary Section may not explain any mathematics.) The relationship could be a matter of historical connection with the subject or with related matters, or of legal, commercial, philosophical, ethical or political position regarding them.

The "**Invariant Sections**" are certain Secondary Sections whose titles are designated, as being those of Invariant Sections, in the notice that says that the Document is released under this License. If a section does not fit the above definition of Secondary then it is not allowed to be designated as Invariant. The Document may contain zero Invariant Sections. If the Document does not identify any Invariant Sections then there are none.

The "**Cover Texts**" are certain short passages of text that are listed, as Front-Cover Texts or Back-Cover Texts, in the notice that says that the Document is released under this License. A Front-Cover Text may be at most 5 words, and a Back-Cover Text may be at most 25 words.

A "**Transparent**" copy of the Document means a machine-readable copy, represented in a format whose specification is available to the general public, that is suitable for revising the document straightforwardly with generic text editors or (for images composed of pixels) generic paint programs or (for drawings) some widely available drawing editor, and that is suitable for input to text formatters or for automatic translation to a variety of formats suitable for input to text formatters. A copy made in an otherwise Transparent file format whose markup, or absence of markup, has been arranged to thwart or discourage subsequent modification by readers is not Transparent. An image format is not Transparent if used for any substantial amount of text. A copy that is not "Transparent" is called "**Opaque**".

Examples of suitable formats for Transparent copies include plain ASCII without markup, Texinfo input format, LaTeX input format, SGML or XML using a publicly available DTD, and standard-conforming simple HTML, PostScript or PDF designed for human modification. Examples of transparent image formats include PNG, XCF and JPG. Opaque formats include proprietary formats that can be read and edited only by proprietary word processors, SGML or XML for which the DTD and/or processing tools are not generally available, and the machine-generated HTML, PostScript or PDF produced by some word processors for output purposes only.

The “**Title Page**” means, for a printed book, the title page itself, plus such following pages as are needed to hold, legibly, the material this License requires to appear in the title page. For works in formats which do not have any title page as such, “Title Page” means the text near the most prominent appearance of the work’s title, preceding the beginning of the body of the text.

A section “**Entitled XYZ**” means a named subunit of the Document whose title either is precisely XYZ or contains XYZ in parentheses following text that translates XYZ in another language. (Here XYZ stands for a specific section name mentioned below, such as “**Acknowledgements**”, “**Dedications**”, “**Endorsements**”, or “**History**”.) To “**Preserve the Title**” of such a section when you modify the Document means that it remains a section “Entitled XYZ” according to this definition.

The Document may include Warranty Disclaimers next to the notice which states that this License applies to the Document. These Warranty Disclaimers are considered to be included by reference in this License, but only as regards disclaiming warranties: any other implication that these Warranty Disclaimers may have is void and has no effect on the meaning of this License.

2. VERBATIM COPYING

You may copy and distribute the Document in any medium, either commercially or noncommercially, provided that this License, the copyright notices, and the license notice saying this License applies to the Document are reproduced in all copies, and that you add no other conditions whatsoever to those of this License. You may not use technical measures to obstruct or control the reading or further copying of the copies you make or distribute. However, you may accept compensation in exchange for copies. If you distribute a large enough number of copies you must also follow the conditions in section 3.

You may also lend copies, under the same conditions stated above, and you may publicly display copies.

3. COPYING IN QUANTITY

If you publish printed copies (or copies in media that commonly have printed covers) of the Document, numbering more than 100, and the Document’s license notice requires Cover Texts, you must enclose the copies in covers that carry, clearly and legibly, all these Cover Texts: Front-Cover Texts on the front cover, and Back-Cover Texts on the back cover. Both covers must also clearly and legibly identify you as the publisher of these copies. The front cover must present the full title with all words of the title equally prominent and visible. You may add other material on the covers in addition. Copying with changes limited to the covers, as long as they preserve the title of the Document and satisfy these conditions, can be treated as verbatim copying in other respects.

If the required texts for either cover are too voluminous to fit legibly, you should put the first ones listed (as many as fit reasonably) on the actual cover, and continue the rest onto adjacent pages.

If you publish or distribute Opaque copies of the Document numbering more than 100, you must either include a machine-readable Transparent copy along with each Opaque copy, or state in or with each Opaque copy a computer-network location from which the general network-using public has access to download using public-standard network protocols a complete Transparent copy of the Document, free of added material. If you use the latter option, you must take reasonably prudent steps, when you begin distribution of Opaque copies in quantity, to ensure that this Transparent copy will remain thus accessible at the stated location until at least one year after the last time you distribute an Opaque copy (directly or through your agents or retailers) of that edition to the public.

It is requested, but not required, that you contact the authors of the Document well before redistributing any large number of copies, to give them a chance to provide you with an updated version of the Document.

4. MODIFICATIONS

You may copy and distribute a Modified Version of the Document under the conditions of sections 2 and 3 above, provided that you release the Modified Version under precisely this License, with the Modified Version filling the role of the Document, thus licensing distribution and modification of the Modified Version to whoever possesses a copy of it. In addition, you must do these things in the Modified Version:

A. Use in the Title Page (and on the covers, if any) a title distinct from that of the Document, and from those of previous versions (which should, if there were any, be listed in the History section of the Document). You may use the same title as a previous version if the original publisher of that version gives permission.

B. List on the Title Page, as authors, one or more persons or entities responsible for authorship of the modifications in the Modified Version, together with at least five of the principal authors of the Document (all of its principal authors, if it has fewer than five), unless they release you from this requirement.

C. State on the Title page the name of the publisher of the Modified Version, as the publisher.

D. Preserve all the copyright notices of the Document.

E. Add an appropriate copyright notice for your modifications adjacent to the other copyright notices.

F. Include, immediately after the copyright notices, a license notice giving the public permission to use the Modified Version under the terms of this License, in the form shown in the Addendum below.

G. Preserve in that license notice the full lists of Invariant Sections and required Cover Texts given in the Document's license notice.

H. Include an unaltered copy of this License.

I. Preserve the section Entitled "History", Preserve its Title, and add to it an item stating at least the title, year, new authors, and publisher of the Modified Version as given on the Title Page. If there is no section Entitled "History" in the Document, create one stating the title, year, authors, and publisher of the Document as given on its Title Page, then add an item describing the Modified Version as stated in the previous sentence.

J. Preserve the network location, if any, given in the Document for public access to a Transparent copy of the Document, and likewise the network locations given in the Document for previous versions it was based on. These may be placed in the "History" section. You may omit a network location for a work that was published at least four years before the Document itself, or if the original publisher of the version it refers to gives permission.

K. For any section Entitled "Acknowledgements" or "Dedications", Preserve the Title of the section, and preserve in the section all the substance and tone of each of the contributor acknowledgements and/or dedications given therein.

L. Preserve all the Invariant Sections of the Document, unaltered in their text and in their titles. Section numbers or the equivalent are not considered part of the section titles.

M. Delete any section Entitled "Endorsements". Such a section may not be included in the Modified Version.

N. Do not retitle any existing section to be Entitled "Endorsements" or to conflict in title with any Invariant Section.

O. Preserve any Warranty Disclaimers.

If the Modified Version includes new front-matter sections or appendices that qualify as Secondary Sections and contain no material copied from the Document, you may at your option designate some or all of these sections as invariant. To do this, add their titles to the list of Invariant Sections in the Modified Version's license notice. These titles must be distinct from any other section titles.

You may add a section Entitled "Endorsements", provided it contains nothing but endorsements of your Modified Version by various parties–for example, statements of peer review or that the text has been approved by an organization as the authoritative definition of a standard.

You may add a passage of up to five words as a Front-Cover Text, and a passage of up to 25 words as a Back-Cover Text, to the end of the list of Cover Texts in the Modified Version. Only one passage of Front-Cover Text and one of Back-Cover Text may be added by (or through arrangements made by) any one entity. If the Document already includes a cover text for the same cover, previously added by you or by arrangement made by the same entity you are acting on behalf of, you may not add another; but you may replace the old one, on explicit permission from the previous publisher that added the old one.

The author(s) and publisher(s) of the Document do not by this License give permission to use their names for publicity for or to assert or imply endorsement of any Modified Version.

5. COMBINING DOCUMENTS

You may combine the Document with other documents released under this License, under the terms defined in section 4 above for modified versions, provided that you include in the combination all of the Invariant Sections of all of the original documents, unmodified, and list them all as Invariant Sections of your combined work in its license notice, and that you preserve all their Warranty Disclaimers.

The combined work need only contain one copy of this License, and multiple identical Invariant Sections may be replaced with a single copy. If there are multiple Invariant Sections with the same name but different contents, make the title of each such section unique by adding at the end of it, in parentheses, the name of the original author or publisher of that section if known, or else a unique number. Make the same adjustment to the section titles in the list of Invariant Sections in the license notice of the combined work.

In the combination, you must combine any sections Entitled “History” in the various original documents, forming one section Entitled “History”; likewise combine any sections Entitled “Acknowledgements”, and any sections Entitled “Dedications”. You must delete all sections Entitled “Endorsements”.

6. COLLECTIONS OF DOCUMENTS

You may make a collection consisting of the Document and other documents released under this License, and replace the individual copies of this License in the various documents with a single copy that is included in the collection, provided that you follow the rules of this License for verbatim copying of each of the documents in all other respects.

You may extract a single document from such a collection, and distribute it individually under this License, provided you insert a copy of this License into the extracted document, and follow this License in all other respects regarding verbatim copying of that document.

7. AGGREGATION WITH INDEPENDENT WORKS

A compilation of the Document or its derivatives with other separate and independent documents or works, in or on a volume of a storage or distribution medium, is called an “aggregate” if the copyright resulting from the compilation is not used to limit the legal rights of the compilation’s users beyond what the individual works permit. When the Document is included in an aggregate, this License does not apply to the other works in the aggregate which are not themselves derivative works of the Document.

If the Cover Text requirement of section 3 is applicable to these copies of the Document, then if the Document is less than one half of the entire aggregate, the Document’s Cover Texts may be placed on covers that bracket the Document within the aggregate, or the electronic equivalent of covers if the Document is in electronic form. Otherwise they must appear on printed covers that bracket the whole aggregate.

8. TRANSLATION

Translation is considered a kind of modification, so you may distribute translations of the Document under the terms of section 4. Replacing Invariant Sections with translations requires special permission from their copyright holders, but you may include translations of some or all Invariant Sections in addition to the original versions of these Invariant Sections. You may include a translation of this License, and all the license notices in the Document, and any Warranty Disclaimers, provided that you also include the original English version of this License and the original versions of those notices and disclaimers. In case of a disagreement between the translation and the original version of this License or a notice or disclaimer, the original version will prevail.

If a section in the Document is Entitled “Acknowledgements”, “Dedications”, or “History”, the requirement (section 4) to Preserve its Title (section 1) will typically require changing the actual title.

9. TERMINATION

You may not copy, modify, sublicense, or distribute the Document except as expressly provided for under this License. Any other attempt to copy, modify, sublicense or distribute the Document is void, and will automatically terminate your rights under this License. However, parties who have received copies, or rights, from you under this License will not have their licenses terminated so long as such parties remain in full compliance.

10. FUTURE REVISIONS OF THIS LICENSE

The Free Software Foundation may publish new, revised versions of the GNU Free Documentation License from time to time. Such new versions will be similar in spirit to the present version, but may differ in detail to address new problems or concerns. See http://www.gnu.org/copyleft/.

Each version of the License is given a distinguishing version number. If the Document specifies that a particular numbered version of this License “or any later version” applies to it, you have the option of following the terms and conditions either of that specified version or of any later version that has been published (not as a draft) by the Free Software Foundation. If the Document does not specify a version number of this License, you may choose any version ever published (not as a draft) by the Free Software Foundation.

ADDENDUM: How to use this License for your documents

To use this License in a document you have written, include a copy of the License in the document and put the following copyright and license notices just after the title page:

> Copyright © YEAR YOUR NAME. Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.2 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover

> Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

If you have Invariant Sections, Front-Cover Texts and Back-Cover Texts, replace the "with . . . Texts." line with this:

> with the Invariant Sections being LIST THEIR TITLES, with the Front-Cover Texts being LIST, and with the Back-Cover Texts being LIST.

If you have Invariant Sections without Cover Texts, or some other combination of the three, merge those two alternatives to suit the situation.

If your document contains nontrivial examples of program code, we recommend releasing these examples in parallel under your choice of free software license, such as the GNU General Public License, to permit their use in free software.