Avram

a virtual machine code interpreter
for avram Version 0.13.0

by Dennis Furey

Short Contents

Table of Contents

Preface

avram is a virtual machine code interpreter. It reads an input file containing a user-supplied application expressed in virtual machine code, and executes it on the host machine. The name is a quasi-acronym for "Applicative ViRtuAl Machine". Notable features are

- strong support for functional programming operations (e.g., list processing)
- interfaces to selected functions from mathematical libraries, such as
  - gsl (numerical integration, differentiation, and series acceleration)
    http://www.gnu.org/software/gsl/
  - mpfr (arbitrary precision arithmetic)
    http://www.mpfr.org
  - minpack (non-linear optimization)
    http://ftp.netlib.org/minpack
  - lapack (linear algebra)
    http://ftp.netlib.org/lapack
  - fftw (fast fourier transforms)
    http://www.fftw.org
  - Rmath (statistical and transcendental functions)
    http://www.r-project.org
  - ufsparse (sparse matrices)
    http://www.cise.ufl.edu/research/sparse/SuiteSparse/current/SuiteSparse/
  - glpk (linear programming by the simplex method)
    http://tech.groups.yahoo.com/group/lp_solve/
  - lpsolve (mixed integer linear programming)
    http://www.llnl.gov/CASC/sundials/
  - kinsol (constrained non-linear optimization)
    http://www.llnl.gov/CASC/sundials/
- interoperability of virtual code applications with other console applications or shells through the expect library
- a simple high-level interface to files, environment variables and command line parameters
- support for various styles of stateless or persistent stream processors (a.k.a. Unix filters)

The reason for writing avram was that I wanted to do some work using a functional programming language, didn't like any functional programming languages that already existed, and felt that it would be less trouble to write a virtual machine emulator than the back end of a compiler. As of version 0.1.0, the first public release of avram as such in 2000, most of the code base had been in heavy use by me for about four years, running very reliably. At this writing some six years later, it has seen even more use with rarely any reliability issues, in some cases attacking large combinatorial problems for weeks or months at a time. These problems have involved both long running continuous execution, and batches of thousands of shorter jobs.

Although the virtual machine is biased toward functional programming, it is officially language agnostic, so `avram` may be useful to anyone involved in the development of compilers for other programming, scripting, or special purpose languages. The crucial advantage of using it in your own project is that rather than troubling over address modes, register allocation, and other hassles inherent in generating native code, your compiler can just dump a fairly high level intermediate code representation of the source text to a file, and let the virtual machine emulator deal with the details. The tradeoff for using a presumably higher level interpreted language is that the performance is unlikely to be competitive with native code, but this issue is mitigated in the case of numerical applications whose heavy lifting is done by the external libraries mentioned above.

Portability is an added bonus. The virtual code is binary compatible across all platforms. Versions of `avram` as of 0.1.0 and later are packaged using GNU autotools and should be possible to build on any platform supporting them. In particular, the package is known to have built successfully on MacOS, FreeBSD, Solaris (thanks to the compile farm at Sourceforge.net) Digital Unix, and Debian GNU/Linux for i386 and Alpha platforms, although it has not been extensively tested on all of them. Earlier versions were compiled and run successfully on Irix and even Windows-NT (with `gcc`).

This document is divided into three main parts, with possibly three different audiences, but they all depend on a basic familiarity with Unix or GNU/Linux systems.

Chapter 1 [User Manual], page 3
: essentially reproduces the information found in the manpage that is distributed with `avram` with a few extra examples and longer explanations. Properly deployed, `avram` should be almost entirely hidden from end users by wrapper scripts, so the "users" to whom this part is relevant would be those involved in preparing these scripts (a matter of choosing the right command line options). Depending on the extent to which this task is automated by a compiler, that may include the compiler writer or the developers of applications.

Chapter 2 [Virtual Machine Specification], page 19
: documents much of what one would need to know in order to write a compiler that generates code executable by `avram`. That includes the complete virtual machine code semantics and file formats. It would also be possible to implement a compatible replacement for `avram` from scratch based on the information in this chapter, in case anyone has anything against C, my coding style, or the GPL. (A few patches to make it `lint` cleanly or a new implementation in good pedagogical Java without pointers would both be instructive exercises. ;-))

Chapter 3 [Library Reference], page 65
: includes documentation on the application program interface and recommended entry points for the C library distributed with `avram`. This information would be of use to those wishing to develop applications incorporating similar features, or to reuse the code for unrelated purposes. It might also be useful to anyone wishing to develop C or C++ applications that read or write data files in the format used by `avram`.

1 User Manual

This chapter provides the basic information on how to use `avram` to execute virtual machine code applications.

`avram` is invoked by typing a command at a shell prompt in one of these three forms.

> `avram` [*general options*]
> `avram` [*filter mode options*] *codefile*[`.avm`]
> `avram` [*parameter mode options*] *codefile*[`.avm`] [*parameters*]

In the second case, `avram` reads from standard input, and may of course appear as part of commands such as

> `avram` [*filter mode options*] *codefile*[`.avm`] `<` *inputfile*
> *anothercommand* `|` `avram` [*filter mode options*] *codefile*[`.avm`]

When `avram` is invoked with the name of an input file (with a default extension *.avm*), it reads virtual machine code from the file and executes it on the host machine.

The virtual code format used by `avram` is designed to support the features of functional or applicative programming languages. Although this chapter documents only the usage of `avram` and not the internals, it will be helpful to keep in mind that the virtual machine code expresses a mathematical function rather than a program in the conventional sense. As such, it performs no action directly, but may be applied in a choice of ways by the user of `avram` according to the precise operation required.

The following sections provide information in greater detail about usage and diagnostics.

1.1 General Options

Regardless of whatever other command line parameters are given, `avram` accepts the following parameters:

`-h, --help`
: Show a summary of options and exit.

`-V,-v, --version`
: Show the version of program and a short copyleft message and exit.

`--emulation=`*version*
: Be backward compatible with an older version of `avram`. This option should include a valid version number, for example *0.13.0*, which is the version of `avram` to be emulated. It can make virtual code applications future proof, assuming that future versions of `avram` correctly support backward compatibility. It may be used in conjunction with any other option in any mode of operation. This copy of the user manual has not been updated since version 0.13.0 of `avram`, so it is unable to document incompatibilities with later versions. The latest version of the manual may be found at `http://www.lsbu.ac.uk/~fureyd/avram`.

`-e, --external-libraries`
: Show a list of libraries with which `avram` has been linked and whose functions therefore could be called from virtual machine programs. This growing list currently includes selected functions from `fftw`, `glpk`, `gsl`, `kinsol`, `lapack`, `minpack`, `mpfr`, `lpsolve`, `Rmath` and `ufsparse` (see [Preface], page 1) which are documented further in Appendix D [External Libraries], page 135.

`-j, --jail`

This option disables execution of shell commands by virtual code applications, which is normally possible by default even for nominally non-interactive applications (see Section 1.4 [Parameter Mode Options], page 7). A virtual code application attempting to spawn a shell (using the `interact` combinator) when this option is selected will encounter an exception rather than successful completion of the operation. This option is provided as a security feature for running untrusted code (see Section 1.7 [Security], page 14), and is incompatible with '`-i`', '`-t`', and '`-s`'.

`-f, --force-text-input`

Normally `avram` will try to guess by looking at a file whether it is an ordinary text file or one that has been written in the virtual code file format, and choose a different internal representation accordingly. An application may require one representation or the other. This option tells `avram` to treat all input files other than the virtual code file (named in the first command line parameter) as text files regardless of whether or not it would be possible to interpret them otherwise. This option may be used in combination with any other option.

1.2 Modes of Operation

Apart from to the capability to print brief help messages and exit, there are two main modes of operation, depending on which options are specified on the command line before the virtual code file name.

For the purpose of choosing the mode of operation, the virtual code filename is taken to be the first command line argument not beginning with a dash. Other conventions relevant to application specific parameters are detailed in Section 1.5 [Command Line Syntax], page 8.

1.2.1 Filter Mode

In filter mode, the argument to the function given by the virtual code is taken from standard input, and the result is written to standard output, except for error messages resulting from a failure to evaluate the function, which are written to standard error. See Section 1.6 [Diagnostics], page 9. Filter mode is indicated whenever these three conditions are all met.

- Either at least one of the filter mode options appears on the command line preceding the first filename parameter, or there are no options at all. See Section 1.3 [Filter Mode Options], page 6.
- Exactly one filename parameter appears on the command line, which is the name of the virtual machine code file.
- Either the filename comes last on the command line, or the '`--unparameterized`' option precedes it, causing everything following it to be ignored.

Examples:

avram mynewapp `<` *inputfilename*

In this example, filter mode is recognized by default because there are no options or input files on the command line to indicate otherwise. (The input file redirected into standard input is not treated by the shell as a command line argument.)

cat somefile | avram -r coolprog > outputfile
: In this example, the '-r' option gives it away, being one of the filter mode options, in addition to the fact that there are no input file parameters or application-specific options.

avram -u devilmaycare.avm --bogusoption ignoredparameter
: In this case, filter mode is forced by the '-u' option despite indications to the contrary.

1.2.2 Parameter Mode

In parameter mode, the argument to the function given by the virtual code is a data structure containing environment variables and command line parameters including files, application specific options, and possibly standard input. The result obtained by evaluating the function is either a data structure representing a set of files to be written, which may include standard output, or a sequence of shell commands to be executed, or a combination of both. Parameter mode is indicated whenever either of these conditions is met.

- Any of the parameter mode options appears on the command line preceding the first filename parameter. See Section 1.4 [Parameter Mode Options], page 7.
- At least one additional filename parameter or option follows the first filename parameter, and the option '--unparameterized' does not precede it.

Examples:

*avram --map-to-each-file prettyprinter.avm *.c *.h --extra-pretty*
: In this example, parameter mode is indicated both by the parameter mode option '--map-to-each-file' and by the presence of input file names and the '--extra-pretty' option. The latter is specific to the hypothetical prettyprinter.avm virtual code application, as indicated by its position on the command line, and is therefore passed to it by avram.

cat ~/specfile | avram reportgenerator -v - /var/log/syslog
: In this example, a hypothetical parameter mode application reportgenerator is able to read '~/specfile' from standard input because of the - used as a parameter.

avram --parameterized grepenv
: In this example, a hypothetical application that searches shell variables is invoked in parameter mode even with no input files or application specific options, because of the '--parameterized' option. Parameter mode invocation is required by the application to give it access to the environment.

avram grepenv --search-targets=PATH,MANPATH
: This example shows an application specific option with both a keyword and a parameter list. They suffice to indicate parameter mode without an explicit '--parameterized' option.

1.3 Filter Mode Options

The options available in filter mode are listed below. Except as otherwise noted, all options are mutually exclusive. Ordinarily a given application will require certain fixed settings of these options and will not work properly if they are set inappropriately.

`-r, --raw-output`
: Normally the result obtained by evaluating the function in the virtual code file must be a list of character strings, which is written as such to standard output. However, if this option is selected, the form of the result is unconstrained, and it will be written in a data file format that is not human readable but can be used by other applications. This option is incompatible with any other options except '`-u`'.

`-c, --choice-of-output`
: When this option is used, the evaluation of the function given by the virtual machine code will be expected to yield a data structure from which `avram` will ascertain whether standard output should be written in text or raw data format. This option should be used only if application is aware of it. It is incompatible with any other options except '`-u`'.

`-l, --line-map`
: Normally the entire contents of standard input up to `EOF` are loaded into memory and used as the argument to the function in the virtual code file. However, this option causes standard input to be read a line at a time, with the function applied individually to each line, and its result in each case written immediately to standard output. A given application either requires this option or does not, and will not work properly in the alternative. This option implies '`--force-text-input`' and is incompatible with any other option except '`-u`'.

`-b, --byte-transducer`
: This option causes standard input to be read one character at a time, evaluating the function given by the virtual code file each time. The function is used as a state transition function that takes a state and input to a next state and output. The output is written concurrently with the input operations. A given application will not work properly with an inappropriate setting of this option. This option implies '`--force-text-input`' and is incompatible with any other option except '`-u`'.

`-u, --unparameterized`
: Normally `avram` guesses whether to use filter mode or parameter mode depending on whether there are any parameters. Selecting this option forces it to operate in filter mode regardless. Any parameters that may appear on the command line after the virtual code file name are ignored. This option may be used in conjunction with any other filter mode option.

1.4 Parameter Mode Options

The parameter mode options are listed below. Except as otherwise noted, any combination of parameter mode options may be selected together, and except as noted, the settings of these options can be varied without breaking the application.

`-q, --quiet`
: `avram` normally informs the user when writing an output file with a short message to standard output. This option suppresses such messages. This option is compatible with any application and any other parameter mode option except '`-a`'.

`-a, --ask-to-overwrite`
: Selecting this option will cause `avram` to ask permission interactively before overwriting an existing file, and to refrain from overwriting it without permission, in which case the contents that were to be written will be lost. This option overrides '`-q`' and is compatible with any other parameter mode option or application.

`-.EXT`
: An option beginning with a dash followed by a period specifies a default extension for input file names. If `avram` doesn't find a file named on the command line, and the filename doesn't already contain a period, `avram` will try to find a file having a similar name but with the default extension appended. The default extension given by this option takes precedence over the hard coded default extensions of *.fun* and *.avm*. At most one default extension can be supplied. This option is compatible with any other parameter mode option and compatible with any application.

`-d, --default-to-stdin`
: If no filename parameter appears on the command line (other than the name of the virtual code file), this option directs `avram` to read the contents of standard input as if it were specified as a command line parameter. (Standard input can also be specified explicitly as a dash. See Section 1.5 [Command Line Syntax], page 8.) This option is compatible with any application and any other parameter mode option except '`-m`'.

`-m, --map-to-each-file`
: Normally `avram` loads the entire contents of all files named on the command line into memory so as to evaluate the virtual machine code application on all of them together. This option can be used to save memory in the case of applications that operate on multiple files independently. It causes `avram` to load only one file at a time and to perform the relevant evaluation and output before loading the next one. Application specific options and standard input (if specified) are read only once and reused. This option is incompatible with '`-d`', and not necessarily compatible with all applications, although some may work both with and without it.

`-i, --interactive`
: This option is used in the case of applications that interact with other programs through shell commands. An application that is meant to be invoked in this

way requires this option and will not work without it, nor will applications that are not of this type work with it. This option is implied by '-t' and '-s', and is compatible with any other parameter mode option.

`-s, --step`
: This option is used in the case of applications that interact with other programs through shell commands, similarly to '-i', and can substitute for it (see above). The option has the additional effect of causing shell commands issued by `avram` on behalf of the application to be written with their results to standard output, and to cause `avram` to pause after displaying each shell command until a key is pressed. This capability may be useful for debugging or auditing purposes but does not otherwise alter the effects of the application. This option is compatible with any other parameter mode option.

`-t, --trace`
: This option is used in the case of applications that interact with other programs through shell commands, but only by way of the `interact` combinator, for which it provides developers a means of low level debugging, particularly deadlock detection. When this option is selected, a verbose trace of all characters exchanged between the functional transducer and the external application are written to standard output, along with some additional control flow diagnostics. This option is compatible with any other parameter mode option.

`-p, --parameterized`
: Normally `avram` tries to guess whether to operate in filter mode or parameter mode based on the options used and the parameters. If there are no parameters and no options, it will default to filter mode, and try to read standard input. However, if this option is selected, it will use parameter mode (and therefore not try to read standard input unless required).

1.5 Command Line Syntax

The command line parameters that follow the virtual code file name when `avram` is used in parameter mode (Section 1.2.2 [Parameter Mode], page 5) are dependent on the specific application. However, all supported applications are constrained for implementation reasons to observe certain uniform conventions regarding their command line parameters, which are documented here to avoid needless duplication.

The shell divides the command line into "arguments" separated by white space. Arguments containing white space or special characters used by the shell must be quoted or protected as usual. File names with wild cards in them are expanded by the shell before `avram` sees them.

`avram` then extracts from the sequence of arguments a sequence of filenames and a sequence of options. Each option consists of a keyword and an optional parameter list. Filenames, keywords, and parameter lists are distinguished according to the following criteria.

1. An argument is treated as a keyword iff it meets these three conditions.
   a. It starts with a dash.

   b. It doesn't contain an equals sign.
   c. It doesn't consist solely of a dash.
2. An argument is treated as a parameter list iff it meets these four conditions.
   a. It doesn't begin with a dash.
   b. It either begins with an equals sign or doesn't contain one.
   c. It immediately follows an argument beginning with a dash, not containing an equals sign and not consisting solely of a dash.
   d. At least one of the following is true.
      1. It doesn't contain a period, tilde, or path separator.
      2. It contains a comma.
      3. It can be interpreted as a C formatted floating point number.
3. An argument is treated as an input file name iff it meets these four conditions.
   a. It doesn't begin with a dash.
   b. It doesn't contain an equals sign.
   c. It doesn't contain a comma.
   d. At least one of the following is true.
      1. It contains a period, tilde, or path separator.
      2. It doesn't immediately follow an argument beginning with a dash, not consisting solely of a dash, and not containing an equals sign.
4. If an argument contains an equals sign but doesn't begin with one, the part on the left of the first equals sign is treated as a keyword and the part on the right is treated as a parameter list.
5. An argument consisting solely of a dash is taken to represent the standard input file.
6. An argument not fitting any of the above classifications is an error.

These conventions are needed for `avram` to detect input file names in a general, position independent way, so that it can preload the files on behalf of the application. Many standard Unix utilities follow these conventions to a large extent, the exceptions being those that employ non-filename arguments without distinguishing syntax, and use positional or other ad hoc methods of command line interpretation. A drop-in replacement for such an application could nevertheless be implemented using `avram` with an appropriate wrapper script, similar to the approach recommended in Section 1.8 [Example Script], page 15, but with suitable keywords inserted prior to the ambiguous arguments.

1.6 Diagnostics

The means exists for virtual code applications to have run time error messages written to standard error on their behalf by `avram`. Any error messages not documented here originate with an application and should be documented by it.

Most error messages originating from `avram` are prefaced by the application name (i.e., the name of the file containing the virtual machine code), but will be prefaced by `avram:` if the error is caused by a problem loading this file itself. Error messages originating from virtual code applications are the responsibility of their respective authors and might not be prefaced by the application name.

The run time errors not specifically raised by the application can be classified as internal errors, i/o errors, overflow errors, file format errors, application programming errors, and configuration related errors.

Some error messages include a code number. The number identifies the specific point in the source code where the condition was detected, for the benefit of the person maintaining it.

1.6.1 Internal Errors

Internal errors should never occur unless the `avram` source code has been carelessly modified, except as noted in Section 1.11 [Bugs], page 16. There are two kinds.

application-name`: virtual machine internal error (code `*nn*`)`
: Most internal errors would be reported by a message of this form if they were to occur. It indicates that some required invariant was not maintained. In such cases, the program terminates immediately, and any results already produced are suspect.

application-name`: `*nn* `unreclaimed` *struct-names*
: A message of this form could be printed at the end of an otherwise successful run. `avram` maintains a count of the number of units allocated for various data structures, and checks that they are all reclaimed eventually as a safeguard against memory leaks. This message indicates that some memory remains unaccounted for.

If a repeatable internal error is discovered, please email a bug report and a small representative test case to the author at `avram-support@basis.uklinux.net`. Include the version number of `avram`, which you can get by running *`avram --version`*.

1.6.2 i/o Errors

These error messages are prefaced with the name of the application. A further explanation as to the reason, obtained from the standard `strerror()` utility, is appended to the messages below if possible.

application-name`: can't read` *filename*
: A file was not able to be opened for reading, typically because it was not found or because the user does not have permission. The file name is displayed with special characters expanded but without any default extensions or search paths that may have been tried. If you think a file exists and should have been found, there may be a problem with your `AVMINPUTS` environment variable (Section 1.10 [Environment], page 16).

application-name`: can't write` *filename*
: A file was not able to be opened for writing.

application-name`: can't write to` *filename*
: A file was successfully opened for writing but became impossible to write thereafter.

application-name: `can't spawn` *command*
: An attempt to execute a shell command on behalf of an interactive application failed during the `exp_popen()` call to the `libexpect` library.

application-name: `can't close` *filename*
: A call to the standard C procedure `fclose()` failed due to unforeseen circumstances. The error is non-fatal but the file should be checked for missing data.

1.6.3 Overflow Errors

These errors are reported by the application name prefacing one of the following messages, except as noted below.

application-name: `counter overflow (code` *nn*`)`
: An overflow occurred in an unsigned long integer being used as a reference counter or something similar. This situation is very unlikely.

application-name: `memory overflow (code` *nn*`)`
: There wasn't enough memory to build an internal data structure. The most likely cause is an attempt to operate on input files that are too large. Standard remedies apply.

The memory overflow or counter overflow messages can also be reported without the application name preface or a code number. In these cases, they arise in the course of evaluating the function given by the application, rather than by loading the input files.

A counter overflow in this case is possible if the application attempts to compute the size of a very large, shared structure using native integer arithmetic.

Memory overflows are possible due to insufficient memory for a valid purpose, but may also occur due to a non-terminating recursion in the virtual machine code. To prevent thrashing or other bad effects from runaway code, the `ulimit` shell command is your friend.

1.6.4 File Format Errors

Certain application crashes result from an application not adhering to the required conventions about data and file formats, or because the application was invoked in the wrong mode (Section 1.2 [Modes of Operation], page 4). These are the following.

application-name: `invalid text format (code` *nn*`)`
: An application that was expected to return a string of characters to be written to a text file returned data that did not correspond to any valid character representation.

application-name: `null character in prompt`
: An interactive application (invoked rightly or wrongly with '`-i`', '`-t`', or '`-s`') is required to exchange strings of non-null characters internally with `avram`, and used a null.

application-name: `invalid file name (code` *nn*`)`
: The data structure representing a file obtained from an application has a name consisting of something other than character strings. This error could be the result of a filter mode application (Section 1.2.1 [Filter Mode], page 4) being invoked in parameter mode. (Section 1.2.2 [Parameter Mode], page 5)

application-name: `null character in file name`
: Similar to the above errors.

application-name: `bad character in file name`
: Slashes, backslashes, and unprintable characters other than spaces are also prohibited in file names.

application-name: `invalid output preamble format`
: According the format used by `avram` for data files, a data file may contain an optional text portion, known as the preamble. This error occurs when a data file obtained from an application can not be written because the preamble is something other than a list of character strings.

application-name: `invalid file specification`
: This error occurs in situations where the data structure for a file obtained by evaluating the application is too broken to permit any more specific diagnosis.

`avram: invalid raw file format in` *application-name*
: The file containing the virtual machine code was not able to be loaded, because the code was not in a recognizable format. Either the file has become corrupted, the compiler that generated it has a bug in it, or the wrong file was used as a virtual code file.

1.6.5 Application Programming Errors

A further class of application crashes results from miscellaneous bugs in the application. These require the application to be debugged and have no user level explanation or workaround, but are listed here for reference. These messages are not normally prefaced by the application name when reported unless the application elects to do so, except for the `invalid profile identifier` message.

- `invalid recursion`
- `invalid comparison`
- `invalid deconstruction`
- `invalid transpose`
- `invalid membership`
- `invalid distribution`
- `invalid concatenation`
- `invalid assignment`
- `unrecognized combinator (code` *nn*`)`
- *application-name*: `invalid profile identifier`
- `unsupported hook`

1.6.6 Configuration Related Errors

The source code distribution of `avram` incorporates a flexible configuration script allowing it to be installed on a variety of platforms. Not all platforms allow support for all features. It is also anticipated that new features may be added to `avram` from time to time. Some problems may therefore occur due to features not being supported at your site for either of these reasons. The following error messages are relevant to these situations.

`unsupported hook`
: If it's not simply due to an application programming error (Section 1.6.5 [Application Programming Errors], page 12) this message may be the result of trying to use an application that requires a newer version of `avram` than the one installed, even though applications should avoid this problem by checking the version number at run time. If this is the reason, the solution would be to install the latest version.

application-name`: I need avram linked with` *foo*`,` *bar* `and` *baz*`.`
: A message of the this form indicates that a new installation may be needed. At this writing (11/11/1), `avram` may report this message with respect to `libexpect5.32`, `tcl8.3`, and `libutil` if any of the '`-i`', '`-t`', or '`-s`' options is used on a system where not all of these libraries were detected when `avram` was installed from a source distribution. (See Section 1.4 [Parameter Mode Options], page 7.) Because `avram` is useful even without interactive applications, these libraries are not considered absolute prerequisites by the configuration script.

`avram: can't emulate version` *version*
: The '`--emulation=`*version*' option obviously won't work if the requested version is newer than the installed version, or if it is not a valid version number (Section 1.1 [General Options], page 3). When that happens, this message is printed instead and `avram` terminates.

`avram: multiple version specifications`
: The '`--emulation=`*version*' option can be used at most once on a command line. This message is printed if it is used more than once. If you only typed it once and got this message, check your aliases and wrapper scripts before reporting a bug.

`avram: unrecognized option:` *option-name*
: may mean that a command line option has been misspelled, or may be another sign of an obsolete version of `avram`. This message will be followed by a usage summary similar to that of the '`--help`' option. (Section 1.1 [General Options], page 3).

application-name`: warning: search paths not supported`
: If the '`argz.h`' header file was not detected during configuration, `avram` will not be able to support search paths in the `AVMINPUTS` environment variable (Section 1.10 [Environment], page 16). This message is a warning that the environment variable is being ignored. If the warning is followed by an i/o error (Section 1.6.2 [i/o Errors], page 10), the latter may be due to a file being in a path that was not searched for this reason. A workaround is to specify the full path names of all input files outside the current working directory. If you don't need search paths, you can get rid of this message by undefining `AVMINPUTS`.

1.6.7 Other Diagnostics and Warnings

`avram: multiple -.EXT options; all but last ignored`
: This message is written when more than one default extension is given as a command line parameter. At most one default extension is allowed. If more than one is given, only the last one is used. The error is non-fatal and `avram` will try to continue. If you need more than one default extension, consider using the hard coded default extensions of '`.fun`' and '`.avm`', or hacking the shell script in which the `avram` command line appears.

application name`: empty operator`
: This message probably means that the virtual code file is corrupt or invalid.

usage summary
: For any errors in usage not covered by other diagnostics, such as incompatible combinations of options, `avram` prints a message to standard error giving a brief summary of options, similar to the output from *`avram --help`*. (See Section 1.1 [General Options], page 3.)

1.7 Security

A few obvious security considerations are relevant to running untrusted virtual code applications. These points are only as reliable as the assumption that the `avram` executable has not been modified to the contrary.

- The applications with the best protection from malicious code are those that run in filter mode, because they have no access to any information not presented to them in standard input, nor the ability to affect anything other than the contents of standard output (provided that the `--jail` command line option is used). The worst they can do is use up a lot of memory, which can be prevented with the `ulimit` command. Unfortunately, not all applications are usable in this mode.
- Parameter mode applications that do not involve the '`-i`', '`-t`' or '`-s`' options are almost as safe (also assuming `--jail`). They have (read-only) access to environment variables, and to the files that are indicated explicitly on the command line. If standard input is one of the files (as indicated by the use of `-` as a parameter), the virtual code application may infer the current date and time. However, a parameter mode application may write any file that the user has permission to write. The '`--ask-to-overwrite`' option should be used for better security, or at least the '`--quiet`' option should not be used. The virtual code can neither override nor detect the use of these options.
- Interactive parameter mode applications (those that use either the '`-i`', '`-t`' or '`-s`' options) are the least secure because they can execute arbitrary shell commands on behalf of the user. This statement also applies to filter mode and parameter mode applications where the '`--jail`' option is not used. Use of '`--step`' is preferable to '`-i`' for making an audit trail of all commands executed, but the application could probably subvert it. The '`--step`' option may be slightly better because it can allow the user to inspect each command and interrupt it if appropriate. However, in most cases a command will not be displayed until it is already executed. Commands executed by

non-interactive applications normally will display no output to that effect. A `chroot` environment may be the only secure way of running untrusted interactive applications.

1.8 Example Script

It is recommended that the application developer (or the compiler) package virtual machine code applications as shell scripts with the `avram` command line embedded in them. This style relieves the user of the need to remember the appropriate virtual machine options for invoking the application, which are always the same for a given application, or even to be aware of the virtual machine at all.

Here is a script that performs a similar operation to the standard Unix `cat` utility.

```
#!/bin/sh
#\
exec avram --force-text-input --default-to-stdin "$0" "$@"
sKYQNTP\
```

That is, it copies the contents of a file whose name is given on the command line to standard output, or copies standard input to standard output if no file name is given. This script can be marked executable (with `chmod`) and run by any user with the directory of the `avram` executable in his or her `PATH` environment variable, even if `avram` had to be installed in a non-standard directory such as '`~/bin`'.

The idea for this script is blatantly lifted from the `wish` manpage. The first line of the script invokes a shell to process what follows. The shell treats the second line as a comment and ignores it. Based on the third line, the shell invokes `avram` with the indicated options, the script itself as the next argument, and whatever command line parameters were initially supplied by the user as the remaining arguments. The rest of the script after that line is never processed by the shell.

When `avram` attempts to load the shell script as a virtual machine code file, which happens as a result of it being executed by the shell, it treats the first line as a comment and ignores it. It also treats the second line as a comment, but takes heed of the trailing backslash, which is interpreted as a comment continuation character. It therefore also treats the third line as a comment and ignores it. Starting with the fourth line, it reads the virtual code, which is in a binary data format encoded with printable characters, and evaluates it.

1.9 Files

`./profile.txt`
: This file is written automatically by `avram` on behalf of applications that include profile annotations. It lists the number of invocations for each annotated part of the application, the total amount of time spent on it (in relative units), the average amount of time for each invocation, and the percentage of time relative to the remainder of the application. The exact format is undocumented and subject to change.

1.10 Environment

An environment variable `AVMINPUTS` can be made to store a list of directories (using the `set` or `export` commands) that `avram` will search for input files. The directories should be separated by colons, similarly to the `PATH` environment variable.

The search paths in `AVMINPUTS` apply only to the names of input files given on the command line (Section 1.5 [Command Line Syntax], page 8) when `avram` is invoked in parameter mode (Section 1.2.2 [Parameter Mode], page 5). They do not apply to the name of the virtual code file, which is always assumed to be either absolute or relative to the current working directory (this assumption being preferable in the case of a script like that of Section 1.8 [Example Script], page 15).

Starting in the first directory in the list of `AVMINPUTS`, `avram` searches for a file exactly as its name appears on the command line (subject to the expansion of special characters by the shell). If it is not found and the name does not contain a period, but a command line option of '`-.EXT`' has been used, `avram` will then search for a file with that name combined with the extension `.EXT`. If '`-.EXT`' has not been used or if no matching file is found with it, `avram` tries the extensions of *`.avm`* and *`.fun`* in that order, provided the given file name contained no periods. If no match is found for any of those cases, `avram` proceeds to search the next directory in the list obtained from `AVMINPUTS`, and so on. It stops searching when the first match is found. For subsequent input files, the search begins again at the first directory.

If `AVMINPUTS` is defined, the current working directory is not searched for input files unless it is listed. If it is empty or not defined, a default list of search paths is used, currently

```
.:/usr/local/lib/avm:/usr/lib/avm:/lib/avm:/opt/avm:/opt/lib/avm\
:/usr/local/share/avm:/usr/share/avm:/share/avm:/opt/avm:/opt/share/avm
```

These paths are defined in `avram.c` and can be changed by recompiling it.

1.11 Bugs

There are no known bugs outstanding, except for any that may be inherent in the external library functions. However, `avram` has been used most extensively on GNU/Linux systems, and the prospect of portability issues with new or lesser used features on other systems can't be excluded.

Though not observed in practice, it's theoretically possible to blow the stack by passing enough functions as arguments to library functions that pass more functions to library functions (e.g., by using nested calls to the gsl integration functions meant for a single variable to evaluate a very high dimensional multiple integral). In all other cases only dynamic heap storage or a constant amount of stack space is used. In particular, this issue is *not* relevant to virtual code applications that don't use external libraries, or that don't pass functions to them as arguments.

`avram` is designed to recover gracefully from memory overflows by always checking for `NULL` results from `malloc()` or otherwise trapping functions that allocate memory. In the event of an overflow, it conveys an appropriate error message to the virtual code application to be handled by the usual exception handling mechanisms. However, there is currently no way for a virtual code application to detect in advance whether sufficient memory is available, nor for it to resume normal operation once an exception occurs. Furthermore, it has been observed on some systems including Irix and 2.4 series Linux kernels that the `avram` process is killed automatically for attempting to allocate too much memory rather than given the chance to recover.

Please send bug reports to `avram-support@basis.uklinux.net`.

2 Virtual Machine Specification

This chapter contains a description of the virtual machine implemented by `avram`, from the point of view of a person wishing to write a compiler that generates code for it. Before reading this chapter, readers should at least skim Chapter 1 [User Manual], page 3 in order to see the big picture. Topics covered in this chapter include data representations, virtual code semantics, and file formats. A toy programming language is introduced for illustrative purposes. The sections in this chapter might not make sense if read out of order the first time through.

2.1 Raw Material

The purpose of this section is to instill some basic concepts about the way information is stored or communicated by the virtual machine, which may be necessary for an understanding of subsequent sections.

The virtual machine represents both programs and data as members of a semantic domain that is straightforward to describe. Lisp users and functional programmers may recognize familiar concepts of atoms and lists in this description. However, these terms are avoided for the moment, in order to keep this presentation self contained and to prevent knowledgeable readers from inferring any unintended meanings.

As a rule, it is preferable to avoid overspecifying any theoretical artifact. In this spirit, the set of entities with which the virtual machine is concerned can be defined purely in terms of the properties we need it to have.

A distinguished element

A particular element of the set is designated, arbitrarily or otherwise, as a distinguished element. Given any element of the set, it is always possible to decide whether or not it is the distinguished element. The set is non-empty and such an element exists.

A binary operator

A map from pairs of elements of the set to elements of the set exists and meets these conditions.

- It associates a *unique* element of the set with any given ordered pair of elements from the set.
- It does not associate the distinguished element with any pair of elements.

For the sake of concreteness, an additional constraint is needed: *the set has no proper subset satisfying the above conditions.* Any number of constructions remain within these criteria, but there is no need to restrict them further, because they are all equivalent for our purposes.

To see that these properties provide all the structure we need for general purpose computation, we may suppose some given set `S` and an operator `cons` having them are fixed, and infer the following points.

- `S` contains at least one element, the distinguished element. Call it `nil`.
- The pair `(nil,nil)` is a pair of elements of `S`, so there must be an element of `S` that `cons` associates with it. We can denote this element `cons(nil,nil)`.

- As no pair of elements is associated with the distinguished element, `cons(nil,nil)` must differ from `nil`, so `S` contains at least two distinct elements.
- The pair `(nil,cons(nil,nil))` therefore differs from `(nil,nil)`, but because it is yet another pair of elements from `S`, there must be an element associated with it by the operator. We can denote this element as `cons(nil,cons(nil,nil))`.
- Inasmuch as the operator associates every pair of elements with a *unique* element, `cons(nil,cons(nil,nil))` must differ from the element associated with any other pair of elements, so it must differ from `cons(nil,nil)`, and we conclude that `nil`, `cons(nil,nil)` and `cons(nil,cons(nil,nil))` constitute three distinct elements of the set `S`.
- By defining `cons(cons(nil,nil),nil)` and `cons(cons(nil,nil),cons(nil,nil))` analogously and following a similar line of reasoning, one may establish the existence of two more distinct elements of `S`.

It is not difficult to see that an argument in more general terms could show that the inclusion of infinitely many elements in `S` is mandated by the properties of the `cons` operator. Furthermore, every element of `S` other than `nil` owes its inclusion to being associated with some other pair of elements by `cons`, because if it were not, its exclusion would permit a proper subset of `S` to meet all of the above conditions. We can conclude that `S` contains exactly `nil` and the countable infinitude of elements of the form `cons(x,y)`, where `x` and `y` are either `nil` or something of the form `cons(...)` themselves.

Some specific examples of sets and operators that have the required properties are as follows.

- the set of natural numbers, with `0` as the distinguished element, and the `cons` operator defined by `cons(`x`,`y`) = ((`x`+`y`)(`x`+`y`+1))/2 + `y` + 1`
- a set of balanced strings of parentheses, with `()` as the distinguished element, and `cons` defined as string concatenation followed by enclosure in parentheses
- a set of ordered binary trees, with the empty tree as the distinguished element, and the `cons` operator as that which takes an ordered pair of trees to the tree having them as its descendents
- a set containing only its own Cartesian product and an arbitrary but fixed element `nil`, with `cons` being the identity function

Each of these models may suggest a different implementation, some of which are more practical than others. The remainder of this document is phrased somewhat imprecisely in terms of a combination of the latter two. The nature of the set in question is not considered further, and elements of the set are described as "trees" or "lists". The distinguished element is denoted by `nil` and the operator by `cons`. Where no ambiguity results, `cons(x,y)` may be written simply as `(x,y)`. These terms should not be seen as constraints on the implementation.

2.2 Concrete Syntax

The previous section has developed a basic vocabulary for statements such as "the virtual machine code for the identity function is `(nil,(nil,nil))`", which are elaborated extensively in the subsequent sections on code and data formats. However, a description in this style would be inadequate without an explanation of how such an entity as `(nil,(nil,nil))` is communicated to `avram` in a virtual machine code file. The purpose of this section is to fill the gap by explaining exactly how any given tree would be transformed to its concrete representation.

The syntax is based on a conversion of the trees to bit strings, followed by grouping the bits into blocks of six, which are then encoded by printable characters. Although anyone is free to modify `avram`, it is recommended that the concrete syntax described here be maintained for the sake of portability of virtual machine code applications.

Building a tree by reading the data from a file requires a more difficult algorithm than the one presented in this section, and is not considered because it's not strictly necessary for a compiler. Procedures for both reading and writing are available to C and C++ users as part of the `avram` library, and are also easily invoked on the virtual code level.

2.2.1 Bit String Encoding

The conversion from trees to bit strings might have been done in several ways, perhaps the most obvious being based on a preorder traversal with each vertex printed as it is traversed. By this method, the entire encoding of the left descendent would precede that of the right in the bit string. This alternative is therefore rejected because it imposes unnecessary serialization on communication.

It is preferable for the encodings of both descendents of a tree to be interleaved to allow concurrent transmission. Although there is presently no distributed implementation of the virtual machine and hence none that takes advantage of this possibility, it is better to plan ahead than to be faced with backward compatibility problems later.

The preferred algorithm for encoding a tree as a bit string employs a queue. The queue contains trees and allows them to be processed in a first-in first-out order. Intuitively, the algorithm works by traversing the tree in level order. To print a tree `T` as a string of `1`s and `0`s, it performs the following steps.

```
Initialize the queue to contain only T
while the queue is not empty do
  if the front element of the queue is nil then
    print 0
  else if the front element of the queue is of the form cons(x,y) then
    print 1
    append x to the back of the queue
    append y to the back of the queue
  end if
  remove the front element of the queue
end while
```

This algorithm presupposes that any given tree `cons(x,y)` can be "deconstructed" to obtain `x` and `y`. The computability of such an operation is assured in theory by the uniqueness property of the `cons` operator, regardless of the representation chosen. If the trees are implemented with pointers in the obvious way, their deconstruction is a trivial constant time operation.

As an example, running the following tree through the above algorithm results in the bit string `1111111010111100100010011000101000101001001 00`.

```
cons(
   cons(
      cons(nil,cons(nil,cons(nil,nil))),
      cons(nil,cons(nil,nil))),
   cons(
      cons(
         cons(nil,cons(nil,cons(nil,cons(nil,nil)))),
         cons(nil,nil)),
      cons(
         cons(
            cons(nil,cons(nil,cons(cons(nil,cons(nil,nil)),nil))),
            cons(nil,nil)),
         nil)))
```

2.2.2 Blocking

After the bit string is obtained as described above, it is grouped into blocks of six. Continuing with the example, the string

```
111111101011110010001001100010100010100100100
```

would be grouped as

```
111111 101011 110010 001001 100010 100010 100100 100
```

Because the number of bits isn't a multiple of six, the last group has to be padded with zeros, to give

```
111111 101011 110010 001001 100010 100010 100100 100000
```

Each of these six bit substrings is then treated as a binary number, with the most significant bit on the left. The numbers expressed in decimal are

```
63 43 50 9 34 34 36 32
```

The character codes for the characters to be written are obtained by adding sixty to each of these numbers, so as to ensure that they will be printable characters. The resulting character codes are

```
123 103 110 69 94 94 96 92
```

which implies that the tree in the example could be written to a file as `{gnE^^‘\`.

2.3 File Format

A virtual code file consists of an optional text preamble, followed by the concrete representation for a tree. The latter uses the syntax described in the previous section. The purpose of this section is to specify the remaining details of the file format.

The format for virtual code files may also be used for other purposes by virtual code applications, as it is automatically detected and parsed by `avram` when used in an input file, and can be automatically written to output files at the discretion of the application.

Other than virtual code files, input files not conforming to this format are not an error as far as `avram` is concerned, because they are assumed to be text files. Applications can detect in virtual code the assumption that is made and report an error if appropriate.

Although the data file format includes no checksums or other explicit methods of error detection, the concrete syntax itself provides a good measure of protection against undetected errors. The probability is vanishingly small that a random alteration to any valid encoding leaves it intact, because every bit in the sequence either mandates or prohibits the occurrence of two more bits somewhere after it. Errors in different parts of the file would have to be consistent with one another to go unnoticed.

2.3.1 Preamble Section

- A file may contain at most one preamble.
- The preamble, if any, is a consecutive sequence of lines beginning with the first line in the file.
- The first line of the preamble must begin with a hash (`#`) character.
- Subsequent lines of the preamble must either begin with a hash, or immediately follow a line that ends with a backslash (`\`) character (or both).

2.3.2 Data Section

- The data or virtual code section of the file begins on the first line of the file that isn't part of the preamble.
- The data section may not contain any hashes, white space, or other extraneous characters other than line breaks.
- If line breaks are ignored, the data section contains a sequence of characters expressing a single tree in the concrete syntax described in Section 2.2 [Concrete Syntax], page 20.

2.4 Representation of Numeric and Textual Data

As noted already, virtual code applications are specified by functions operating on elements of a set having the properties described in Section 2.1 [Raw Material], page 19, which are convenient to envision as ordered binary trees or pairs of `nil`. However, virtual code applications normally deal with numeric or textual data, for example when they refer to the contents of a text file. It is therefore necessary for the application and the virtual machine emulator to agree on a way of describing textual or numeric data with these trees.

The purpose of this section is to explain the basic data structures used in the exchange of information between `avram` and a virtual code application. For example, an explanation is needed for statements like "an application invoked with the '`--baz`' option is expected to return a pair `(`*foo*`,`*bar*`)`, where *foo* is a list of character strings ...", that are made subsequently in this document. Such statements should be understood as referring to the trees representing the pairs, lists, character strings, etc., according to the conventions explained below.

Characters

An arbitrarily chosen set of 256 trees is used to represent the character set. They are listed in Appendix A [Character Table], page 123. For example, the letter `A` is represented by `(nil,(((nil,(nil,(nil,nil))),nil),(nil,nil)))`. That means that when an application wants the letter `A` written to a text file, it returns something with this tree in it.

Booleans The value of `false` is represented by `nil`, and the value of `true` is represented by `(nil,nil)`.

Pairs Given any two items of data *x1* and *x2*, having the respective representations *r1* and *r2*, the pair (*x1*,*x2*) has the representation cons(*r1*,*r2*).

Lists A list of the items *x1*, *x2* ... *xn* with respective representations *r1* through *rn* is represented by the tree cons(*r1*,cons(*r2*...cons(*rn*,nil)...)). In other words, lists are represented as pairs whose left sides are the heads and whose right sides are the tails. The empty list is identified with `nil`. Lists of arbitrary finite length can be accommodated.

Naturals A number of the form *b0* + 2*b1* + 4*b2* + ... + 2^n *bn*, where each *b*i is `0` or `1`, is represented by a tree of the form cons(*t0*,cons(*t1*...cons(*tn*,nil)...)) where each *t*i is `nil` if the corresponding *b*i is `0`, and `(nil,nil)` otherwise. Note that the numbers *b*i are exactly the bits written in the binary expansion of the number, with *b0* being the least significant bit.

Strings are represented as lists of characters.

`avram` imposes no more of a "type discipline" than necessary to a workable interface between it and an application. This selection of types and constructors should not be seen as constraining what a compiler writer may wish to have in a source language.

2.5 Filter Mode Interface

From the point of view of the application developer or compiler writer, there are parameter mode applications, which are discussed in Section 2.6 [Parameter Mode Interface], page 27, and filter mode applications, which are discussed in this section. Of the latter, there are mainly three kinds: those that read one character at a time, those that read a line at a time, and those that read the whole standard input file at once. Each of them is invoked with different options and expected to follow different calling conventions. This section summarizes these conventions.

2.5.1 Loading All of Standard Input at Once

Unless '`--line-map`' or '`--byte-transducer`' is used as a command line option when the application is invoked, the contents of standard input are loaded entirely into memory by `avram` before evaluation of the virtual code begins. This interface is obviously not appropriate for infinite streams.

The virtual code application in this mode of operation is treated as a single function taking the entire contents of standard input as its argument, and returning the entire contents of standard output as its result. Hence, this interface is one of the simplest available.

2.5.1.1 Standard Input Representation

The representation for the standard input file used as the argument to the function depends both on the file format and on the command line options specified when the application is invoked. The '`--unparameterized`' and '`--raw-output`' options make no difference to the input representation, and the '`--line-map`' and '`--byte-transducer`' options are not relevant to this mode of operation. That leaves four possible combined settings of the '`--choice-of-output`' and '`--force-text-input`' options. If standard input conforms to the data file format specification Section 2.3 [File Format], page 22, the following effects are possible.

- If neither '`--choice-of-output`' nor '`--force-text-input`' is used, the argument to the function will be given directly by the tree encoded in the data section of the file. The preamble of the file will be ignored.
- If the '`--choice-of-output`' option is used and the '`--force-text-input`' option is not used, the argument to the function will be a pair (*preamble*,*contents*), where *preamble* is a list of character strings taken from the preamble of the file (with leading hashes stripped), and *contents* is the tree represented in the data section of the file.
- If the '`--choice-of-output`' option is not used and the '`--force-text-input`' option is used, the argument to the function will be the whole file as a list of character strings. I.e., both the preamble and the data sections are included, hashes are not stripped from the preamble, and the data section is not converted to the tree it represents.
- If the '`--choice-of-output`' option is used and the '`--force-text-input`' option is also used, the argument to the the function will be a pair (`nil`,*contents*), where the contents are the list of character strings as in the previous case.

If standard input does not conform to the data file format specification in Section 2.3 [File Format], page 22, then it is assumed to be a text file. The '`--force-text-input`' option makes no difference, and there are only two possible effects, depending on whether '`--choice-of-output`' is used. They correspond to the latter two cases above, where '`--force-text-input`' is used.

The idea of the '`--choice-of-output`' option is that it is used only for applications that are smart enough to be aware of the (*preamble*,*contents*) convention. A non-empty preamble implies a data file whose contents could be any type, but an empty preamble implies a text file whose contents can only be a list of character strings. (In the case of a data file with no preamble, the list of the empty string is used for the preamble to distinguish it from a text file.)

Dumb applications that never want to deal with anything but text files should be invoked with '`--force-text-input`'. Otherwise, they have to be prepared for either text or data as arguments.

The use of both options at once is unproductive as far as the input format is concerned, but may be justified when the output is to be a data file and the input is text only.

2.5.1.2 Standard Output Representation

As in the case of standard input, the representation for standard output that the function is expected to return depends on the command line options with which the application is invoked. The only relevant options are '`--raw-output`' and '`--choice-of-output`', which are mutually exclusive.

- If neither option is selected, the result returned by the function must be a list of character strings.
- If '`--raw-output`' is used, the result returned by the function is unconstrained, and it will be written as a data file with no preamble, following the format specified in Section 2.3 [File Format], page 22.
- If '`--choice-of-output`' is used, the result returned by the function must be a pair `(`*preamble*`,`*contents*`)`.

In the last case, the preamble determines how the file will be written. If it is meant to be a text file, the preamble should be `nil`, and the contents should be a list of character strings. If it is meant to be a data file, the preamble should be a non-empty list of character strings, and the format of the contents is unconstrained. To express a data file with no preamble, the preamble should be the list containing the empty string, rather than being empty.

In the result returned by the function, the preamble lines should not include leading hash characters, because they are automatically added to the output to enforce consistency with the data file format. However, they should include trailing backslashes as continuation characters where appropriate. The hashes that are automatically added will be automatically stripped by `avram` on behalf of whatever application uses the file.

Any file can be written as a list of character strings, even "text" files that are full of unprintable characters, and even "text" files that happen to conform to the format used for data files. However, if the application intends to write a data file in the standard format used by other virtual code applications, it can do so more quickly and easily by having the virtual machine do the formatting automatically with the '`--choice-of-output`' option than by implementing the algorithm in Section 2.2 [Concrete Syntax], page 20, from scratch in virtual code.

2.5.2 Line Maps

Virtual code applications invoked with the '`--line-map`' option (with or without the '`--unparameterized`' option) adhere to a very simple interface.

- The argument to the function is a character string, and the result must also be a character string.
- The function is applied to each line of the standard input file and the result in each case is written to standard output followed by a line break.

This kind of application may be used on finite or infinite streams, provided that the lengths of the lines are finite, but preserves no state information from one line to the next.

2.5.3 Byte Transducers

The interface used when the `--byte-transducer` option is selected allows an application to serve as a persistent stream processor suitable for finite or infinite streams. The interface can be summarized by the following points.

- When it is first invoked, the function in the virtual code file is applied to an argument of `nil`, and is expected to return a pair `(`*state*`,`*output*`)`. The *state* format is unconstrained. The *output* must be a character string that will be written to standard output, but it may be the empty string.
- For each byte read from standard input, `avram` applies the function to the pair `(`*state*`,`*character*`)`, using the state obtained from previous evaluation, and the character whose code is the byte. The purpose of the *state* field is therefore to provide a way for the application to remember something from one invocation to the next.
- The function is usually expected to return a pair `(`*state*`,`*output*`)` for each input byte, so that the state can be used on the next iteration, and the output can be written to standard output as a character string.
- If the function ever returns a value of `nil`, the computation terminates.
- If standard input comes to an end before the computation terminates, the function will be applied to a pair of the form `(`*state*`,nil)` thereafter, but may continue to return `(`*state*`,`*output*`)` pairs for arbitrarily many more iterations. The `EOF` character is not explicitly passed to the function, but the end is detectable insofar as `nil` is not a representation for any character.

Unlike the situation with line maps, the output character strings do not have line breaks automatically appended, and the application must include them explicitly if required. The convention for line breaks is system dependent. On Unix and GNU/Linux systems, character code 10 indicates a line break, but other systems may use character code 13 followed by character code 10. See Appendix A [Character Table], page 123 for the representations of characters having these codes.

2.6 Parameter Mode Interface

The virtual code file for a parameter mode application contains a tree representing a single function, which takes as its argument a data structure in the format described below. The format of the result returned by the function depends on the virtual machine options used on the command line, and the various alternatives are explained subsequently.

2.6.1 Input Data Structure

The data structure that is used as the argument to the parameter mode application incorporates all the information about the command line and the environment variables. It is in the form of a triple `((`*files*`,`*options*`),`*environs*`)`. The fields have these interpretations.

files is a list of quadruples `((`*date*`,`*path*`),(`*preamble*`,`*contents*`))`, with one quadruple for each input file named on the command line (but not the virtual code file or the `avram` executable). The list will be in the same order as the filenames on the command line, and is not affected by options interspersed with them. The fields in each item have the following interpretations.

date is the time stamp of the file in as a character string in the usual Unix format, for example, `Fri Jan 19 14:34:44 GMT 2001`. If the file corresponds to standard input, the current system time and date are used.

path is the full path of the file, expressed as a list of strings. If the file corresponds to standard input, the list is empty. Otherwise, the first string in the list is the file name. The next is the name of the file's parent directory, if any. The next is the parent of the parent, and so on. The root directory is indicated by the empty string, so that any path ending with the empty string is an absolute path, while any path ending with a non-empty string is relative to the current working directory. Path separators (i.e., slashes) are omitted.

preamble In the case of a text file, or any file not conforming to the format in Section 2.3 [File Format], page 22, this field is `nil`. Otherwise, this field contains the preamble of the file expressed as a list of strings, or contains just the empty string if the file has no preamble. Any leading hashes in the preamble of the file are stripped.

contents In the case of a text file (as indicated by the *preamble* field), this field will contain a list of character strings, with each line of the file contained in a character string. Otherwise, it can contain data in any format, which are obtained by converting the data section of the file to a tree.

options is a list of quadruples of the form `((`*position*`,`*longform*`),(`*keyword*`,`*params*`))` with one quadruple for each option appearing on the command line after the name of the virtual code file.

position is a natural number indicating the position of the option on the command line. The position numbers of all the options will form an ascending sequence, but not necessarily consecutive nor starting with zero. The missing numbers in the sequence will correspond to the positions of the file names on the command line, allowing their positions to be inferred by applications for which the position matters.

longform is a boolean value which is true if the option starts with two or more dashes but false otherwise.

keyword is the key word of the option expressed as a character string. For example in the case of a command line option `--foo=bar,baz`, the keyword is `foo`. Leading dashes are stripped.

params is a list of character strings identifying the parameters for the command line option in question. In the case of an option of the form `--foo=bar,baz`, the first character string in the list will be `bar` and the next will be `baz`. The same applies if the option is written `--foo bar,baz` or `--foo =bar,baz`. If there are no parameters associated with the option, the list is empty.

environs is a list of pairs of character strings, with one pair in the list for each environment variable. The identifier is the left string in the pair, and the value is the right. For example, if the environment contains the definition `OSTYPE=linux-gnu`, there will be a pair in the list whose left side is the string `OSTYPE` and whose right side is the string `linux-gnu`.

2.6.2 Input for Mapped Applications

Applications invoked using the '`--map-to-each-file`' option benefit from a slightly different interface than the one described above. As the purpose of this option is to save memory by loading only one file at a time, the application does not have access to all input files named on the command line simultaneously within the same data structure. Although the data structure is of the type already described, the *files* field is not a list of arbitrary length. Instead, it is a list containing exactly one item for an input file. If `-` is used as a command line parameter, indicating standard input, then *files* will have another item pertaining to standard input. In no event will it have other than one or two items.

The mapped application is expected to work by being applied individually to each of any number of separately constructed data structures, doing the same in each case as it would if that case were the only one. To make that possible, copies of the environment variables, the contents of standard input, and the list of application specific options are contained in the data structure used for every invocation.

The position numbers in the options are adjusted for each invocation to reflect the position of the particular input file associated with it. For example, in the following command

```
avram --map-to-each-file mapster.avm fa.txt --data fb.dat --code fc.o
```

the function in the virtual code file '`mapster.avm`' would be applied to each of three data structures, corresponding to the commands

```
avram mapster.avm fa.txt --data --code
avram mapster.avm --data fb.dat --code
avram mapster.avm --data --code fc.o
```

If the relative positions of the options and filenames were important to the application, they could be reliably inferred from the position numbers. In the first case, they would be 1 and 2, implying that the file is in position 0. In the second case they would be 0 and 2, implying that the file is in position 1, and in the third case they would be 0 and 1, implying that the file is in position 2. (Of course, nothing compels an application to concern itself with the positions of its parameters, and the alternative might be preferable.)

For the most part, any application that can operate on one file at a time without needing information from any others can be executed more economically with the '`--map-to-each-file`' option and few if any changes to the code. The effect will normally be analogous to the above example, subject to a few possible differences.

- If an application is supposed to do something by default when there are no file parameters or only standard input, it won't work as a mapped application, because if there are no file parameters it won't be executed at all.
- If a mapped application causes any output files to be generated, they may be written before other input files are read, possibly causing the input files to be overwritten if they have the same names, and causing subsequent invocations to use the overwritten versions. This behavior differs from that of loading all input files at the outset, which ensures the application seeing all of the original versions. The latter may be more convenient for maintaining a group of files in some sort of consistent state.

- If an application causes standard output to be written along with output files, normally standard output is written last as a security measure against malicious code altering the '`--ask-to-overwrite`' prompts by subtly clobbering the console. In a mapped application, standard output isn't always last because there may be more invocations to come.

2.6.3 Output From Non-interactive Applications

If a parameter mode application is not invoked with either of the '`--interactive`' or '`--step`' options, then it is deemed to be non-interactive, and therefore does not concern itself with executing shell commands. Instead, it simply specifies a list of output files to be created or updated on its behalf by `avram`.

The files are described by a list of quadruples `((`*overwrite*`,`*path*`),(`*preamble*`,`*contents*`))`, with one quadruple for each file.

In each quadruple, the *path*, *preamble*, and *contents* fields have the same interpretations as in the list of files in the input data structure described in Section 2.6.1 [Input Data Structure], page 27, except that a `nil` path refers to standard output rather than to standard input.

The *overwrite* field in each quadruple tells whether the file should be overwritten or appended. If the *overwrite* field is `nil` (i.e., the representation for the Boolean value of `false`) and a file already exists at the given path, the new contents will be appended to it. If the *overwrite* field is anything other than `nil` and/or no file exists with the given path, a new file is written or the extant one is overwritten. Note that the data file format specified in Section 2.3 [File Format], page 22 precludes appending anything to it, but the format of existing output files is not checked and nothing prevents data or text from being appended to one.

2.6.4 Output From Interactive Applications

Parameter mode applications invoked with either of the '`--interactive`' or '`--step`' options are required to take the data structure described in Section 2.6.1 [Input Data Structure], page 27 as an argument but to return the virtual code for a function that will observe a certain protocol allowing shell commands to be executed on its behalf. The intent is that the virtual code file doesn't contain the real application *per se*, but only something that builds the real one on the fly using configuration information from the input files and command line options.

The format of the result returned by an interactive application, being a virtual code application itself, requires a full exposition of the virtual machine code semantics. This subject is deferred to Section 2.7 [Virtual Code Semantics], page 33. The remainder of this section describes the protocol followed by the function returned by the interactive application rather than the application itself.

Similarly to the case of a byte transducer described in Section 2.5.3 [Byte Transducers], page 26, the basic pattern of interaction between `avram` and the function is a cycle of invocations. In general terms, the function is applied to a `nil` argument initially, and expected to return an initial state and initial output. Thereafter, the function is applied to a pair of the state returned on the previous iteration, and the next installment of input. The function returns further output and a new state, and the cycle continues until the function returns a value of `nil`, at which time the computation terminates.

2.6.4.1 Line Oriented Interaction

Within this general pattern, more specific styles of interaction are possible. In the simplest one to explain first, the result returned by the function is always a data structure of the form (*state*,(*command lines*,*prompts*)), wherein the fields have these interpretations.

state is a tree incorporating any data in any format that the application needs to remember from one invocation to the next.

command lines
is a list of character strings that are piped to the standard input stream of a separately spawned process. The process may persist from one invocation of the function to the next, or may be spawned each time.

prompts is a non-empty list of character strings containing a suffix of the text expected from the standard output stream of the process as a result of sending the command lines to it.

On each iteration, `avram` sends the command line character strings to a separately spawned process, with line breaks between them if there are more than one command. If a process remains from the previous iteration that has not terminated itself, the list of command lines is sent to the same process. If no such process already exists, the first string in the list of command lines is treated as a shell command and used to spawn the process (using the `exp_popen` library function), and the remaining strings are sent to the newly spawned process.

Normally processes spawned with commands that invoke interactive command line interpreters of their own, such as `bash`, `ftp` or `bc`, will persist indefinitely unless the command causing them to exit is issued or some other event kills them. Processes spawned with non-interactive commands, such as `ls` or `pwd`, will terminate when the last of their initial output has been received.

In the case of processes that persist after being spawned, `avram` needs some way of knowing when to stop waiting for more output from them so that it doesn't get stuck waiting forever. This purpose is served by the *prompts* field. This field could contain a single string holding the last thing the process will send before becoming quiescent, such as the strings `bash$` or `ftp>` in the above examples. Alternatively, a sequence of more than one prompt string can be used to indicate that the corresponding sequence of lines must be detected. An empty string followed by `ftp>` would indicate that the `ftp>` prompt is expected to be immediately preceded by a line break. There is also the option of using prompt strings to indicate a pattern that does not necessarily imply quiescence, but is a more convenient point at which to stop reading the output from the process.

For processes spawned with commands that do not start their own interactive command line interpreters, such as `ls` or `pwd`, it may be preferable to read all the output from them until they terminate. To achieve this effect, the list of prompt strings should contain only the single string containing only the single `EOF` character (usually code 4) or any other character that is certain not to occur in the output of the process. This technique is based on the assumption that the process was spawned originally with the command in question, not that such a command is sent to an existing shell process.

In any case, when enough output has been received from the process, it is collected into a list of received strings including the prompt strings at the end (if they were received), and the function is applied to the pair `(`*state*`,`*received strings*`)`. If the cycle is to continue, the result returned by the function will include a new state, a new list of command lines, and a new list of prompt strings. A result of `nil` will cause the computation to terminate.

There are some unusual situations that could occur in the course of line oriented interaction, and are summarized as follows.

- If the process terminates before any pattern matching the prompt strings is received from it, all of the output from the process up to the point where it terminated is collected into the *received strings* list and passed to the function. This situation includes cases where the process terminates immediately upon being spawned, but not abnormal completion of the `exp_popen` library function, which is a fatal error. This feature of the interface is what allows `EOF` to be used for collecting all the output at once from a non-interactive command.
- If the list of *command lines* is empty, and no process currently exists due to a previous iteration, the effect is the same as if the process terminates unexpectedly before outputting anything. I.e., the function is applied to a pair containing an empty list of received strings. There is no good reason for an application to get into this situation.
- If the list of *command lines* is empty but a process persists from a previous iteration, no output is sent to it, but receiving from it proceeds normally. This feature of the interface could be used effectively by applications intended to process the received data in parts, but will cause deadlock if the process is already quiescent.
- All character strings have to consist of lists of valid representations of non-null characters according to Appendix A [Character Table], page 123, or else there will be some fatal error messages.
- If the list of *prompt strings* contains only the empty string, `avram` will not wait to receive anything from the process, but will proceed with the next iteration immediately. If this effect is intended, care must be taken not to confuse the empty list of *prompt strings* with the list containing the empty string. The former indicates character oriented interaction, which is explained next.

2.6.4.2 Character Oriented Interaction

A character oriented style of interaction involves the function always returning a data structure of the form `(`*state*`,(`*command lines*`,nil))`. The *state* and *command lines* fields serve exactly the same purposes respectively as they do in the case of line oriented interaction. The field that would be occupied by the *prompt strings* list in the case of line oriented interaction is identically `nil` in this style.

When this style is used, `avram` spawns a process and/or sends command lines to it as in the case of line oriented interaction, but attempts to read only a single character from it per iteration. When the character is received, `avram` applies the function to the pair `(`*state*`,`*character*`)` in order to obtain the next state and the next list of command lines. If the process has terminated, a `nil` value is used in place of the character. If the process is quiescent, deadlock ensues.

The character oriented style is a lower level protocol that shifts more of the burden of analyzing the process's output to the virtual code application. It can do anything line oriented interaction can do except proceeding immediately without waiting to receive any output from the process. It may also allow more general criteria (in effect) than the matching of a fixed prompt string to delimit the received data, for those pathological processes that may require such things.

Applications using character oriented interaction need to deal with line breaks explicitly among the received characters, unlike the case with line oriented interaction, where the line breaks are implicit in the list of received strings. Contrary to the convention for Unix text files, line breaks in the output of a process are indicated by character code 13 followed by character code 10.

2.6.4.3 Mixed Modes of Interaction

An application is not confined exclusively to line oriented or character oriented interaction, but may switch from one style to the other between iterations, and signal its choice simply by the format of the data structure it returns. If the *prompt strings* field is non-empty, the interaction is line oriented, and if the field is empty, the interaction is character oriented. A function using both styles has to be prepared for whichever type of data it indicates, either a character or a list of character strings as the case may be.

Another alternative is possible if the function returns a data structure in the form `(`*files*`,nil)`. This structure includes neither a list of command lines nor a list of prompt strings, empty or otherwise, but does include a list of quadruples in the *files* field. The quadruples are of the form `((`*overwrite*`,`*path*`),(`*preamble*`,`*contents*`))`. The fields have the same interpretations as in the output from a non-interactive parameter mode application, as described in Section 2.6.3 [Output From Non-interactive Applications], page 30, and will cause a list of files to be written in the same way.

As an interactive application is able cause the execution of arbitrary shell commands, it doesn't need `avram` to write files for it the way a non-interactive application does, so this feature does not provide any additional capabilities. However, it may be helpful as a matter of convenience.

After the files are written, the function will be applied to the same result it returned, `(`*files*`,nil)`. There is no direct means of preserving unconstrained state information from previous iterations in this style of interaction. A likely scenario might therefore be that the function returns a file list after finishing its other business, and then returns `nil` on the next iteration to terminate.

2.7 Virtual Code Semantics

As the previous sections explain, virtual code applications are defined in terms of mathematical functions. Up until this point, the discussion has focused on the interface between the function and the virtual machine interpreter, by detailing the arguments passed to the functions under various circumstances and the results they are expected to return in order to achieve various effects.

The purpose of this section is to complete the picture by explaining how a given computable function can be expressed in virtual code, considering only functions operating on the trees described in Section 2.1 [Raw Material], page 19. Functions manipulating trees of `nil` are undoubtedly a frivolous and abstract subject in themselves. One is obliged to refer back to the previous sections if in need of motivation.

2.7.1 A New Operator

With regard to a set of trees as described in Section 2.1 [Raw Material], page 19, we can define a new binary operator. Unlike the `cons` operator, this one is not required to associate an element of the set with every possible pair of elements. For very many pairs of operands we will have nothing to say about its result. In fact, we require nothing of it beyond a few simple properties to be described presently.

Because this is the only other operator than `cons`, there is no need to have a special notation for it, so it will be denoted by empty space. The tree associated by the operator with a pair of trees x and y, if any, will be expressed in the infix notation $x\ y$. For convenience, the operator is regarded as being right associative, so that $a\ b\ c$ can be written for a ($b\ c$).

2.7.2 On Equality

One example of a property this operator should have, for reasons that will not be immediately clear, is that for any trees x and k, the equality `cons(cons(nil,`k`),nil)` x = k always holds. Even though the present exposition opts for readability over formality, statements like these demand clarification of the notion of equality. Some of the more pedantic points in Section 2.1 [Raw Material], page 19 may be needed for the following ideas to hold water.

- As originally stipulated, it is always possible to distinguish `nil` from any member of the set. We can therefore decide on this basis whether a = b whenever at least one of them is `nil`.
- Where neither a nor b is `nil` in an expression a = b, but rather something of the form `cons(`x`,`y`)`, the equality holds if and only if both pairs of corresponding subexpressions are equal. If at least one member of each pair of corresponding subexpressions is `nil`, the question is settled, but otherwise there is recourse to their respective subexpressions, and so on. This condition follows from the uniqueness property of the `cons` operator.
- If one side of an equality is of the form $x\ y$, or is defined in terms of such an expression, but (x,y) is one of those pairs with which the operator associates no result, then the equality holds if and only if the other side is similarly ill defined.
- Statements involving universal quantification (i.e., beginning with words similar to "for any tree x . . .") obviously do not apply to instances of a variable (x) outside the indicated set (trees). Hence, they are not refuted by any consequence of identifying a variable with an undefined expression.

Readers who are aware of such issues as pointer equality or intensional versus extensional equality of functions are urged to forget all about them in the context of this document, and abide only by what is stated. Other readers should ignore this paragraph.

2.7.3 A Minimal Set of Properties

For any trees x, y, and k, and any non-`nil` trees p, f, and g, the new invisible operator satisfies these conditions. In these expressions and hereafter, increasing abuse of notation is perpetrated by not writing the `cons` in expressions of the form `cons(`x`,`y`)`.

P0 `(nil,(nil,nil))` $x = x$

P1 `(nil,((nil,nil),nil))` `(`x`,`y`)` $= x$

P2 `(nil,(nil,(nil,nil)))` `(`x`,`y`)` $= y$

P3 `((nil,`k`),nil)` $x = k$

P4 `(((nil,(nil,nil)),nil),nil)` `(`f`,`x`)` $= f$ `(`f`,`x`)`

P5 `((`f`,`g`),nil)` $x = f\ g\ x$

P6 `((`f`,nil),`g`)` $x =$ `(`$f\ x$`,`$g\ x$`)`

P7 `((`p`,`f`),`g`)` $x = f\ x$ if $p\ x$ is a non-`nil` tree, but $g\ x$ if $p\ x =$ `nil`

Although other properties remain to be described, it is worth pausing at this point because there is ample food for thought in the ones already given. An obvious question would be that of their origin. The short answer is that they have been chosen arbitrarily to be true by definition of the operator. At best, the completion of the construction may lead to a more satisfactory answer based on aesthetic or engineering grounds.

A more important question would be that of the relevance of the mystery operator and its properties to the stated purpose of this section, which is to specify the virtual machine code semantics. The answer lies in that the operator induces a function for any given tree t, such that the value returned by the function when given an argument x is $t\ x$. This function is the one that is implemented by the virtual code t, which is to say the way an application will behave if we put t in its virtual code file. An equivalent way of looking at the situation is that the virtual machine does nothing but compute the result of this operator, taking the tree in the virtual code file as its left operand and the input data as the right operand. By knowing what the operator will do with a given pair of operands, we know what to put into the virtual code file to get the function we want.

It is worthwhile to note that properties *P0* to *P7* are sufficient for universality in the sense of Turing equivalence. That means that any computable function could be implemented by the suitable choice of a tree t without recourse to any other properties of the operator. A compiler writer who finds this material boring could therefore stop reading at this point and carry out the task of targeting any general purpose programming language to the virtual machine based on the specifications already given. However, such an implementation would not take advantage of the features for list processing, exception handling, or profiling that are also built into the virtual machine and have yet to be described.

2.7.4 A Simple Lisp Like Language

With a universal computational model already at our disposal, it will be easier to use the virtual machine to specify itself than to define all of it from scratch. For this purpose, we use the `silly` programming language, whose name is an acronym for SImple Lisp-like Language (Yeah right). The language serves essentially as a thin layer of symbolic names on top of the virtual machine code. Due to its poor support for modularity and abstraction, `silly` is not recommended for serious application development, but at least it has a shallow learning curve.[1]

2.7.4.1 Syntax

`silly` has no reserved words, but it has equals signs, commas and parentheses built in. A concise but ambiguous grammar for it can be given as follows.

program ::= *declaration**

declaration ::= *identifier* `=` *expression*

expression ::= `()` | *identifier* | `(`*expression*`)` | `(`*expression*`,`*expression*`)`
| *expression expression* | *expression*`(`*expression*`)`
| *expression*`(`*expression*`,`*expression*`)`

The real grammar is consistent with this one but enforces right associativity for binary operations and higher precedence for juxtaposition without intervening white space.

The declaration of any identifier must be unique and must precede its occurrence in any expression. Hence, cyclic dependences between declarations and "recursive" declarations are not allowed.

2.7.4.2 Semantics

Declarations in `silly` should be understood in the obvious way as preprocessor directives to perform parenthetic textual substitutions (similar to `#define id (exp)` in C). All identifiers in expressions are thereby eliminated during the preprocessing phase.

The overall meaning of the program is the meaning of the expression in the last declaration. A denotational semantics for expressions is given by the following equations, where [[x]] should be read as "the meaning of x", and x, y and z are metavariables. (That is, they stand for any source code fragment that could fit there subject to the constraint, informally speaking, that it has to correspond to a connected subtree of the parse tree as enforced by the unambiguous grammar in the context of the rest of the program.)

$$[[\texttt{()}]] = \texttt{nil}$$

$$[[\texttt{(}x\texttt{)}]] = [[x]]$$

$$[[\texttt{(}x\texttt{,}y\texttt{)}]] = \texttt{cons(}[[x]]\texttt{,}[[y]]\texttt{)}$$

$$[[x\ \ y]] = [[x\texttt{(}y\texttt{)}]] = [[x]]\ [[y]]$$

[1] Previous releases of `avram` included a working `silly` compiler, but this has now been superseded by the Ursala programming language. Ursala includes `silly` as a subset for the most part, and the examples in this manual should compile and execute with very little modification.

$$[[x\ \ (y,z)]] = [[x(y,z)]] = [[x]]\ [[(y,z)]]$$

Toy languages like this are among the few situations a denotational semantics stands a chance of clarifying more than it obfuscates, so the reader is invited to take a moment to savor it.

2.7.4.3 Standard Library

`silly` programs may be linked with library modules, which consist of `silly` source text to be concatenated with the user program prior to the preprocessing phase. Most `silly` programs are linked with the standard `silly` prelude, which contains the following declarations among others.

```
nil          = ()
identity     = (nil,(nil,nil))
left         = (nil,((nil,nil),nil))
right        = (nil,(nil,(nil,nil)))
meta         = (((nil,(nil,nil)),nil),nil)
constant_nil = ((nil,nil),nil)
couple       = ((((left,nil),constant_nil),nil),right)
compose      = couple(identity,constant_nil)
constant     = couple(couple(constant_nil,identity),constant_nil)
conditional  =
   couple(couple(left,compose(left,right)),compose(right,right))
```

There is a close correspondence between these declarations and the properties described in Section 2.7.3 [A Minimal Set of Properties], page 35. A fitting analogy would be that the properties of the operator specify the virtual machine instruction set in a language independent way, and the `silly` library defines the instruction mnemonics for a virtual assembly language. The relationship of some mnemonics to their corresponding instructions may be less clear than that of others, so they are all discussed next.

2.7.5 How `avram` Thinks

The definitions in the standard `silly` library pertaining to the basic properties of the operator can provide a good intuitive illustration of how computations are performed by `avram`. This task is approached in the guise of a few trivial correctness proofs about them. Conveniently, as an infeasibly small language, `silly` is an ideal candidate for analysis by formal methods.

Technically the semantic function $[[\ldots]]$ has not been defined on identifiers, but we can easily extend it to them by stipulating that the meaning of an identifier x is the meaning of the program *main* = x when linked with a library containing the declaration of x, where *main* is an identifier not appearing elsewhere in the library.

With this idea in mind, the following "theorems" can be stated, all of which have a similar proof. The variables x and y stand for any tree, and the variable f stands for any tree other than `nil`.

T0 $[\![$`identity`$]\!]\ x = x$

T1 $[\![$`left`$]\!]$ `(`x`,`y`)` $= x$

T2 $[\![$`right`$]\!]$ `(`x`,`y`)` $= y$

T4 $[\![$`meta`$]\!]$ `(`f`,`x`)` $= f$ `(`f`,`x`)`

T5 $[\![$`constant_nil`$]\!]\ x =$ `nil`

Replacing each identifier with its defining expression directly demonstrates a logical equivalence between the relevant theorem and one of the basic operator properties postulated in Section 2.7.3 [A Minimal Set of Properties], page 35.

For more of a challenge, it is possible to prove the next theorem.

T6 For non-`nil` f and g, ($[\![$`couple`$]\!]$ `(`f`,`g`)`) $x =$ `(`f x`,`g x`)`

The proof is a routine calculation. Beware of the distinction between the mathematical `nil` and the `silly` identifier `nil`.

($[\![$`couple`$]\!]$ `(`f`,`g`)`) $x =$ ($[\![$`((((left,nil),constant_nil),nil),right)`$]\!]$ `(`f`,`g`)`) x

by substitution of `couple` with its definition in the standard library

$=$ `((((`$[\![$`left`$]\!]$`,`$[\![$`nil`$]\!]$`),`$[\![$`constant_nil`$]\!]$`),`$[\![$`nil`$]\!]$`),`$[\![$`right`$]\!]$`)` `(`f`,`g`)`) x

by definition of the semantic function $[\![\ldots]\!]$ regarding pairs

$=$ `((((`$[\![$`()`$]\!]$ ...

by substitution of `nil` from its definition in the standard library

$=$ `((((`$[\![$`left`$]\!]$`,nil),`$[\![$`constant_nil`$]\!]$`),nil),`$[\![$`right`$]\!]$`)` `(`f`,`g`)`) x

by definition of the semantic function in the case of $[\![$`()`$]\!]$

$=$ `((`$[\![$`left`$]\!]$ `(`f`,`g`),`$[\![$`constant_nil`$]\!]$ `(`f`,`g`)),`$[\![$`right`$]\!]$ `(`f`,`g`))` x

by property *P6* (twice)

$=$ `((`f`,nil),`g`)` x

by theorems *T1*, *T2*, and *T5*

$=$ `(`f x`,`g x`)`

by property *P6* again.

Something to observe about this proof is that it might just as well have been done automatically. Every step is either the substitution of an identifier or a pattern match against existing theorems and properties of the operator. Another thing to note is that the use of identifiers and previously established theorems helps to make the proof human readable, but is not a logical necessity. An equivalent proof could have been expressed entirely in terms of the properties of the operator. If one envisions a proof like this being performed blindly and mechanically, without the running commentary or other amenities, that would not be a bad way of thinking about what takes place when `avram` executes virtual code.

Three more theorems have similar proofs. For non-`nil` trees p, f and g, and any trees x and k:

T7 $([[\texttt{compose}]]\ \texttt{(}f\texttt{,}g\texttt{)})\ x = f\ g\ x$

T8 $([[\texttt{constant}]]\ k)\ x = k$

T9 $([[\texttt{conditional}]]\ \texttt{(}p\texttt{,(}f\texttt{,}g\texttt{))}\ x = f\ x$ if $p\ x$ is non-`nil`, but $g\ x$ if $p\ x = \texttt{nil}$

The proofs of these theorems are routine calculations analogous to the proof of *T6*. Here is a proof of theorem *T7* for good measure.

$$([[\texttt{compose}]]\ \texttt{(}f\texttt{,}g\texttt{)})\ x = ([[\texttt{couple(identity,constant\_nil)}]]\ \texttt{(}f\texttt{,}g\texttt{)})\ x$$

by substitution of `compose` with its definition in the standard library

$$= (\texttt{(}[[\texttt{couple}]]\ \texttt{(}[[\texttt{identity}]]\texttt{,}[[\texttt{constant\_nil}]]\texttt{))(}f\texttt{,}g\texttt{)})\ x$$

by definition of the semantic function

$$= \texttt{(}[[\texttt{identity}]]\ \texttt{(}f\texttt{,}g\texttt{),}[[\texttt{constant\_nil}]]\ \texttt{(}f\texttt{,}g\texttt{))}\ x$$

by theorem *T6*

$$= \texttt{((}f\texttt{,}g\texttt{),nil)}\ x$$

by theorems *T0* and *T5*

$$= f\ g\ x$$

by property *P5* of the operator.

2.7.6 Variable Freedom

The virtual code semantics is easier to specify using the `silly` language than it would be without it, but still awkward in some cases. An example is the following declaration from the standard library,

```
hired = compose(
   compose,
   couple(
      constant compose,
      compose(couple,couple(constant,constant couple))))
```

which is constructed in such a way as to imply the following theorem, provable by routine computation.

T9 $([[\texttt{hired}]]\ h)\ \texttt{(}f\texttt{,}g\texttt{)} = [[\texttt{compose}]]\texttt{(}h\texttt{,}[[\texttt{couple}]]\texttt{(}f\texttt{,}g\texttt{))}$

Intuitively, `hired` represents a function that takes a given function to a higher order function. For example, if `f` were a function that adds two real numbers, `hired f` would be a function that takes two real valued functions to their pointwise sum.

Apart from its cleverness, such an opaque way of defining a function has little to recommend it. The statement of the theorem about the function is more readable than the function definition itself, probably because the theorem liberally employs mathematical variables, whereas the `silly` language is variable free. On the other hand, it is not worthwhile to linger over further enhancements to the language, such as adding variables to it. The solution will be to indicate informally a general method of inferring a variable free function definition from an expression containing variables, and hereafter omit the more cumbersome definitions.

An algorithm called `isolate` does the job. The input to `isolate` is a pair `(`e`,`x`)`, where e is a syntactically correct `silly` expression in which the identifier x may occur, but no other identifiers dependent on x may occur (or else it's garbage-in/garbage-out). Output is a syntactically correct `silly` expression f in which the identifier x does not occur, such that $[[e]] = [[f\ x]]$. The algorithm is as follows,

```
if e = x then
  return identity
else if e is of the form (u,v)
  return couple(isolate(u,x),isolate(v,x))
else if e is of the form u v
  return (hired apply)(isolate(u,x),isolate(v,x))
else
  return constant e
```

where equality is by literal comparison of expressions, and the definition of `apply` is

`apply = (hired meta)((hired compose)(left,constant right),right)`

which represents a function that does the same thing as the invisible operator.

T10 $[[$`apply`$]]$ `(`f`,`x`)` $= f\ x$

The `isolate` algorithm can be generalized to functions of arbitrarily many variables, but in this document we will need only a unary and a binary version. The latter takes an expression e and a pair of identifiers `(`x`,`y`)` as input, and returns an expression f such that $[[e]] = [[f$ `(`x`,`y`)`$]]$.

```
if e = x then
  return left
else if e = y then
  return right
else if e is of the form (u,v)
  return couple(isolate(u,(x,y)),isolate(v,(x,y)))
else if e is of the form u v
  return (hired apply)(isolate(u,(x,y)),isolate(v,(x,y)))
else
  return constant e
```

It might be noted in passing that something similar to these algorithms would be needed in a compiler targeted to `avram` if the source were a functional language with variables.

2.7.7 Metrics and Maintenance

Certain features of the virtual machine pertain to software development and maintenance more than to implementing any particular function. The operations with the mnemonics `version`, `note`, `profile`, and `weight` are in this category.

2.7.7.1 Version

A virtual code application with exactly the following definition implements a function that returns a constant character string regardless of its argument.

```
version = ((nil,nil),((nil,nil),(nil,((nil,nil),nil))))
```

The character string encodes the version number of the installed `avram` executable, for example `0.13.0`, using the standard representation for characters.

Although such an application is useless by itself, the intended use for this feature is to cope with the possibility that future versions of `avram` may include enhancements. Ideally, the maintainer of `avram` will update the version number when new enhancements are added. Applications can then detect whether they are available in the installed version by using this feature. If a needed enhancement is not available, the application can either make allowances or at least terminate gracefully.

2.7.7.2 Note

This operation allows arbitrary information or comments to be embedded in a virtual code application in such a way that it will be ignored by `avram` when executing it. For the `silly` language, a `note` function is defined in the standard prelude so as to imply the following theorem.

T11 $[[\texttt{note}]]\ (f,c) = \texttt{((nil,nil),((nil,nil),(nil,(nil,(}f\texttt{,}c\texttt{)))))}$

Intuitively, the argument f represents a function, and the argument `c` represents the comment, annotation, or whatever, that will be embedded but ignored in the virtual code.

Semantically, a function with a note attached is the same as the function by itself, as the following property stipulates for any non-`nil` f.

P8 $([[\texttt{note}]]\ (f,c))\ x = f\ x$

A possible reason for using this feature might be to support a language that performs run-time type checking by hanging type tags on everything. Another possible use would be to include symbolic information needed by a debugger.

2.7.7.3 Profile

The virtual machine supports a profiling capability by way of this feature. Profiling an application causes run time statistics about it to be written to a file '`./profile.txt`'. Profiled applications are of the form indicated in the following theorem

T12 $[[\texttt{profile}]]\ (f,s) = \texttt{((nil,nil),((nil,nil),(nil,((}f\texttt{,}s\texttt{),nil))))}$

where f stands for the virtual code of the application, and s stands for the name of it to be written to the file. The semantics of a profiled function is identical to the unprofiled form for any non-`nil` f.

P9 ([[`profile`]] (f,s)) $x = f\ x$

Unlike the situation with `note`, the annotation s of used in profiled code is not in an unrestricted format but must be a character string in the standard representation (as in Section 2.4 [Representation of Numeric and Textual Data], page 23), because this string needs to be written by `avram` to the file '`./profile.txt`'. Ordinarily this string will be the source code identifier of the function being profiled.

When profiles are used in many parts of an application, an informative table is generated showing the time spent in each part.

2.7.7.4 Weight

The following virtual code implements a function that returns the weight of its argument in the standard representation for natural numbers.

```
weight = ((nil,nil),((nil,nil),(nil,(nil,nil))))
```

The weight of a tree is zero if the tree is `nil`, and otherwise the sum of the weights of the two subtrees plus one.

An algorithm to compute the weight of a tree would be trivial to implement without being built in to the virtual machine. The built in capability could also be used for purposes unrelated to code maintenance. Nevertheless, it is built in for the following reasons.

- Computing weights happened to be a bottleneck for a particular aspect of code generation that was of interest to the author, namely the compression of generated code.
- A built in implementation in C runs at least an order of magnitude faster than the equivalent implementation in virtual code.
- It runs even faster when repeated on the same data, by retrieving previously calculated weights rather than recalculating them.

The only disadvantage of using this feature instead of implementing a function in virtual code to compute weights is that it relies on native integer arithmetic and could overflow, causing a fatal error. It has never occurred in practice, but is possible due to sharing, whereby the nominal weight of a tree could be exponentially larger than the actual amount of memory occupied by it.

2.7.8 Deconstruction

Much of the time required for evaluating a function is devoted to performing deconstruction operations, e.g., taking the left side of a pair, the tail of a list, the right side of the head of the tail, etc.. Because these operations are so frequent, there are some features of the virtual machine to make them as efficient as possible.

2.7.8.1 Field

The virtual machine supports a generalization of the `left` and `right` deconstruction operations that is applicable to deeply nested structures. Use of this feature is conducive to code that is faster and more compact than is possible by relying on the primitive deconstructors alone. It may also be easier for a code optimizer to recognize and transform.

The general form of a virtual code application to perform deconstruction is that it is a pair with a `nil` left side, and a non-`nil` right side. The right side indicates the nature of the deconstruction to be performed when the function is evaluated on an argument.

To make the expression of deconstruction functions more readable in `silly`, the standard library contains the declaration

```
field = couple(constant nil,identity)
```

which implies the following theorem.

T13 $[[\texttt{field}]]\ w = (\texttt{nil},w)$

The virtual machine recognizes an application in this form and evaluates it according to the following properties, where u and v are other than `nil`, but x, y, and z are unrestricted.

P10 $([[\texttt{field}]]\ (u,\texttt{nil}))\ (x,y) = ([[\texttt{field}]]\ u)\ x$

P11 $([[\texttt{field}]]\ (\texttt{nil},v))\ (x,y) = ([[\texttt{field}]]\ v)\ y$

P12 $([[\texttt{field}]]\ (u,\texttt{v}))\ z = (([[\texttt{field}]]\ u)\ z,([[\texttt{field}]]\ v)\ z)$

One might also add that $([[\texttt{field}]]\ (\texttt{nil},\texttt{nil}))\ z = z$, but this statement would be equivalent to *P0*.

A suitable choice of the `field` operand permits the implementation of any deconstruction function expressible in terms of `compose`, `couple`, `identity`, `left` and `right`. For example, the application `couple(compose(right,right),left)` has an equivalent representation in `field((nil,(nil,(nil,nil))),((nil,nil),nil))`. The latter looks longer in `silly` but is smaller and faster in virtual code.

2.7.8.2 Fan

In cases where a deconstructions would be needed to apply the same function to both sides of a pair, the overhead can be avoided by means of a property of the virtual machine intended for that purpose.

A `silly` definition of `fan` implying the following theorem is helpful in expressing such an application.

T14 $[[\texttt{fan}]]\ f = ((\texttt{nil},\texttt{nil}),((\texttt{nil},f),(\texttt{nil},\texttt{nil})))$

The virtual machine recognizes when an application has the form shown above, and uses f as a function to be applied to both sides of the argument.

P13 $([[\texttt{fan}]]\ f)\ (x,y) = (f\ x,f\ y)$

2.7.9 Recursion

Defining functions or programs self referentially is sometimes informally known as recursion. In functional languages, the clever use of “combinators” is often preferred to this practice, and is in fact well supported by the virtual machine. However, some computations may be inexpressible without an explicitly “recursive” formulation, so there is some support for that as well.

2.7.9.1 Recur

A generalization of the form denoted by `meta` in `silly` is recognized by the virtual machine and allows a slightly more efficient encoding of recursive applications. An expression `recur` p has the representation indicated by this theorem,

T15 $\quad [[\texttt{recur}]]\ p = \texttt{(((nil,}p\texttt{),nil),nil)}$

which implies that $[[\texttt{meta}]] = [[\texttt{recur}]]\ \texttt{(nil,nil)}$.

If p is non-`nil`, a tree of the form $[[\texttt{recur}]]\ p$ is interpreted as follows. Note that *P4* is equivalent to the special case of this property for which p is `(nil,nil)`.

P14 $\quad ([[\texttt{recur}]]\ p)\ x = [[\texttt{meta}]]\ ([[\texttt{field}]]\ p)\ x$

The rationale is that `meta` would very frequently be composed with a deconstruction `field` p, so the virtual machine saves some time and space by allowing the two of them to be encoded in a smaller tree with the combined meaning.

2.7.9.2 Refer

In the style of recursive programming compelled by the available `meta` primitive, a function effectively requires a copy of its own machine code as its left argument. Bringing about that state of affairs is an interesting party trick.

If we had a definition of `bu` in the standard library implying

T16 $\quad ([[\texttt{bu}]]\ \texttt{(}f\texttt{,}k\texttt{)})\ x = f\texttt{(}k\texttt{,}x\texttt{)}$

which for the sake of concreteness can be done like this,

```
bu = (hired compose)(
        left,
        (hired couple)(compose(constant,right),constant identity))
```

then a definition of `refer` as

```
refer = (hired bu)(identity,identity)
```

would be consistent with the following property of the operator.

P15 $\quad ([[\texttt{refer}]]\ f)\ x = f\ \texttt{(}f\texttt{,}x\texttt{)}$

The proof, as always, is a matter of routine calculation in the manner of the section on how `avram` thinks.

However, this pattern would occur so frequently in recursively defined functions as to be a significant waste of space and time. Therefore, rather than requiring it to be defined in terms of other operations, the virtual machine specification recognizes a pattern of the form below with respect to property *P15*,

T17 $\quad [[\texttt{refer}]]\ f = \texttt{(((}f\texttt{,nil),nil),nil)}$

and takes the property to be true by definition of the operator. A definition of `refer` consistent with *T17* is therefore to be found in the standard library, not the definition proposed above.

2.7.10 Assignment

In an imperative programming paradigm, a machine consists partly of an ensemble of addressable storage locations, whose contents are changed over time by assignment statements. An assignment statement includes some computable function of the global machine state, and the address of the location whose contents will be overwritten with the value computed from the function when it is evaluated.

Compiling a language containing assignment statements into virtual machine code suitable for `avram` might be facilitated by exploiting the following property.

P16 ([[`assign`]] `(`p`,`f`)`) x = [[`replace`]] `((`p`,`f x`),`x`)`

The identifier `assign` is used in `silly` to express a virtual code fragment having the form shown below, and `replace` corresponds to a further operation to be explained presently.

T18 [[`assign`]] `(`p`,`f`)` = `(((`p`,`f`),nil),nil)`

This feature simulates assignment statements in the following way. The variable x in *P16* corresponds intuitively to the set of addressable locations in the machine. The variable f corresponds to the function whose value will be stored in the location addressed by p. The result of a function expressed using `assign` is a new store similar to the argument x, but with the part of it in location p replaced by f x. A source text with a sequence of assignment statements could therefore be translated directly into a functional composition of trees in this form.

The way storage locations are modeled in virtual code using this feature would be as nested pairs, and the address p of a location is a tree interpreted similarly to the trees used as operands to the `field` operator described in Section 2.7.8.1 [Field], page 42, to specify deconstructions. In fact, `replace` can be defined as a minimal solution to the following equation.

E0 ([[`field`]] p) [[`replace`]] `((`p`,`y`),`x`)` = y

This equation regrettably does not lend itself to inferring the `silly` source for `replace` using the `isolate` algorithm in Section 2.7.6 [Variable Freedom], page 39, so an explicit construction is given in Section B.3 [Replace], page 130. This construction need not concern a reader who considers the equation a sufficiently precise specification in itself.

In view of the way addresses for deconstruction are represented as trees, it would be entirely correct to infer from this equation that a tuple of values computed together can be assigned to a tuple of locations. The locations don't even have to be "contiguous", but could be anywhere in the tree representing the store, and the function is computed from the contents of all of them prior to the update. Hence, this simulation of assignment fails to capture the full inconvenience of imperative programming except in the special case of a single value assigned to a single location, but fortunately this case is the only one most languages allow.

There is another benefit to this feature besides running languages with assignment statements in them, which is the support of abstract or opaque data structures. A function that takes an abstract data structure as an argument and returns something of the same type can be coded in a way that is independent of the fields it doesn't use. For example, a data structure with three fields having the field identifiers `foo`, `bar`, and `baz` in some source language might be represented as a tuple `((`*foo contents*`,`*bar contents*`),`*baz contents*`)` on the virtual code level. Compile time constants like `bar = ((nil,(nil,nil)),nil)` could be defined in an effort to hide the details of the representation, so that the virtual code `field bar` is used instead of `compose(right,left)`. Using field identifiers appropriately, a function that transforms such a structure by operating on the `bar` field could have the virtual code `couple(couple(field foo,compose(f,field bar)),field baz)`. However, this code does not avoid depending on the representation of the data structure, because it relies on the assumption of the `foo` field being on the left of the left, and the `baz` field being on the right. On the other hand, the code `assign(bar,compose(f,field bar))` does the same job without depending on anything but the position of the `bar` field. Furthermore, if this position were to change relative to the others, the code maintenance would be limited to a recompilation.

2.7.11 Predicates

A couple of operations are built into the virtual machine for performing tests efficiently. These functions return either `nil` for false or `(nil,nil)` for true, and are useful for example as a predicate p in programs of the form `conditional(`p`,(`f`,`g`))` among other things. In this example, the predicate is applied to the argument, a result of `(nil,nil)` causes f to be applied to it, and a result of `nil` causes g to be applied to it.

2.7.11.1 Compare

A function that performs comparison has a the following very simple virtual code representation.

T19 [[`compare`]] = `(nil,nil)`

The proof of theorem *T19* is that the standard `silly` prelude contains the declaration `compare = (nil,nil)`. Code in this form has the following semantics.

P17 For distinct trees x and y, [[`compare`]] `(`x`,`y`)` = `nil`

P18 [[`compare`]] `(`x`,`x`)` = `(nil,nil)`

In other words, the virtual code `(nil,nil)` implements a function that takes a pair of trees and returns true if and only if they are equal.

It would be fairly simple to write an equivalent virtual code application that implements this function if it were not realizable in this form by definition of the operator. However, this method is preferable because it saves space in virtual code and has a highly optimized implementation in C.

2.7.11.2 Member

Another built in predicate function has the virtual code shown below.

T20 [[`member`]] = `((nil,nil),((nil,nil),nil))`

As the mnemonic suggests, the function implemented by this code detects whether a given item is a member of a list. The left side of its argument is the item to be detected, and the right side is the list that may or may not contain it. Lists are represented as explained in Section 2.4 [Representation of Numeric and Textual Data], page 23.

The virtual code semantics can be specified by these three properties of the operator.

P19 $[\![\texttt{member}]\!]$ $(x,\texttt{nil}) = \texttt{nil}$

P20 $[\![\texttt{member}]\!]$ $(x,(x,y)) = (\texttt{nil},\texttt{nil})$

P21 For distinct trees x and y, $[\![\texttt{member}]\!]$ $(x,(y,z)) = [\![\texttt{member}]\!]$ $(x,\texttt{z})$

As in the case of `compare`, the implementation of `member` is well optimized in C, so this form is to be preferred over an ad hoc construction of a membership testing function in virtual code.

2.7.12 Iteration

One of many alternatives to recursion provided by the virtual machine is iteration, which allows an operation to be repeated until a condition is met. If the source language is imperative, this feature provides an easy means of translating loop statements to virtual code. In languages that allow functions to be treated as data, iteration can be regarded as a function that takes a predicate and a function as arguments, and returns a function that applies the given function repeatedly to its argument until the predicate is refuted.

Iterative applications are expressed in virtual code by the pattern shown below.

T21 $[\![\texttt{iterate}]\!]$ $(p,f) = ((\texttt{nil},\texttt{nil}),(\texttt{nil},(p,f)))$

In the `silly` language, the `iterate` mnemonic plays the role of the function that takes the virtual code for a predicate p and a function f as arguments, and returns the virtual code for an iterating function.

The code for an iterating function is recognized as such by the virtual machine emulator only if the subtrees f and p are other than `nil`. The resulting function tests the argument x with p and returns x if the predicate is false.

P22 $([\![\texttt{iterate}]\!]$ $(p,f))$ $x = x$ if p $x = \texttt{nil}$

If the predicate turns out to be true, f is applied to x, and then another iteration is performed.

P23 $([\![\texttt{iterate}]\!]$ $(p,f))$ $x = ([\![\texttt{iterate}]\!]$ $(p,f))$ f x if p x is a non-`nil` tree

2.7.13 List Combinators

There is extensive support for operations on lists in the virtual code format. Use of these features is encouraged because they are conducive to tight code with explicit concurrency. Within an imperative programming paradigm, these features might perhaps have to be understood as design patterns or algorithmic skeletons. The present exposition takes a functional view, describing them in terms of operators that take functions as their arguments and return functions as their result.

2.7.13.1 Map

A virtual code application in the following form causes a function with non-`nil` virtual code f to be applied to every item in a list.

T22 $[\![\texttt{map}]\!]$ $f = ((\texttt{nil},\texttt{nil}),((\texttt{nil},f),\texttt{nil}))$

The `map` mnemonic is used in `silly` to express applications in this form as `map` f. This operation is also well known to lisp users and functional programmers. The semantics is determined by these two operator properties (for non-`nil` f).

P24 $([[\texttt{map}]]\ f)\ \texttt{nil} = \texttt{nil}$

P25 $([[\texttt{map}]]\ f)\ \texttt{(}x\texttt{,}y\texttt{)} = \texttt{(}f\ x\texttt{,(}[[\texttt{map}]]\ f\texttt{)}\ y\texttt{)}$

Note that the representation of lists described in Section 2.4 [Representation of Numeric and Textual Data], page 23, is assumed.

2.7.13.2 Filter

Another well known list operation is that which applies a predicate to every item of a list, and deletes those for which the predicate is false. For a predicate with virtual code p, such an application can be coded conveniently in this form,

T23 $[[\texttt{filter}]]\ p = \texttt{((nil,nil),(nil,(}p\texttt{,nil)))}$

which is to say that writing `((nil,nil),(nil,(`p`,nil)))` in `silly` is the same as writing `filter` p.

The virtual machine detects code of this form provided that p is other than `nil`, and evaluates it consistently with the following properties, causing it to have the meaning that it does.

P26 $([[\texttt{filter}]]\ p)\ \texttt{nil} = \texttt{nil}$

P27 $([[\texttt{filter}]]\ p)\ \texttt{(}x\texttt{,}y\texttt{)} = ([[\texttt{filter}]]\ p)\ y$ if $p\ x = \texttt{nil}$

P28 $([[\texttt{filter}]]\ p)\ \texttt{(}x\texttt{,}y\texttt{)} = \texttt{(}x\texttt{,(}[[\texttt{filter}]]\ p\texttt{)}\ y\texttt{)}$ if $p\ x$ is a non-`nil` tree

2.7.13.3 Reduce

In the virtual code fragment shown below, f should be regarded as the virtual code for a binary operator, and k is a constant.

T24 $[[\texttt{reduce}]]\ \texttt{(}f\texttt{,}k\texttt{)} = \texttt{((nil,nil),((}f\texttt{,}k\texttt{),nil))}$

By constructing a tree in the form shown, the `silly` mnemonic `reduce` effectively constructs the code for a function operating on lists in a particular way.

The effect of evaluating an application in this form with an argument representing a list can be broken down into several cases.

- If the list is empty, then the value of k is the result.
- If the list has only one item, then that item is the result.
- if the list has exactly two items, the first being x and the second being y, then the result is f `(`x`,`y`)`.
- If the list has more than two items and an even number of them, the result is that of evaluating the application with the list of values obtained by partitioning the list into pairs of adjacent items, and evaluating f with each pair.
- If the list has more than two items and an odd number of them, the result is that of evaluating the application with the list of values obtained by partitioning the list into pairs of adjacent items excluding the last one, evaluating f with each pair, and then appending the last item to the list of values.

In the last two cases, evaluation of the application is not necessarily finished after just one traversal of the list, because it has to carry on until the list is reduced to a single item.

Some care has been taken to describe this operation in detail because it differs from comparable operations common to functional programming languages, variously known as fold or reduce. All of these operations could be used, for example, to compute the summation of a list of numbers. The crucial differences are as follows.

- Whereas a fold or a reduce is conventionally either of the left or right variety, this `reduce` is neither.
- The vacuous case result k is never used at all unless the argument is the empty list.
- This `reduce` is not very useful if the operator f is not associative.

The reason for defining the semantics of `reduce` in this way instead of the normal way is that a distributed implementation of this one could work in logarithmic time, so it's worth making it easy for a language processor to detect the pattern in case the virtual code is ever going to be targeted to such an implementation. Of course, nothing prevents the conventional left or right reduction semantics from being translated to virtual code by explicit recursion.

The precise semantics of this operation are as follows, where f is not `nil`, k is unconstrained, and `pairwise` represents a function to be explained presently.

P29 ([[`reduce`]] `(`f`,`k`)`) `nil` = k

P30 ([[`reduce`]] `(`f`,`k`)`) `(`x`,`y`)` =
[[`left`]] ([[`bu(iterate,right)`]] [[`pairwise`]] f) `(`x`,`y`)`

The latter property leverages off some virtual machine features and `silly` code that has been defined already. The function implemented by [[`pairwise`]] f is the one that partitions its argument into pairs and evaluates f with each pair as described above. The combination of that with `bu(iterate,right)` results in an application that repeatedly performs [[`pairwise`]] f while the resulting list still has a tail (i.e., a `right` side), and stops when the list has only a single item (i.e., when `right` is false). The `left` operation then extracts the item.

For the sake of completeness, it is tedious but straightforward to give an exact definition for `pairwise`. The short version is that it can be anything that satisfies these three equations.

E1 ([[`pairwise`]] f) `nil` = `nil`

E2 ([[`pairwise`]] f) `(`x`,nil)` = `(`x`,nil)`

E3 ([[`pairwise`]] f) `(`x`,(`y`,`z`))` = `(`f `(`x`,`y`)`,([[`pairwise`]] f) z`)`

For the long version, see Section B.1 [Pairwise], page 129.

2.7.13.4 Sort

Sorting is a frequently used operation that has the following standard representation in virtual code.

T25 [[`sort`]] p = `((nil,nil),((`p`,nil),(nil,nil)))`

When an application in this form is evaluated with an operand representing a list, the result is a sorted version of the list.

The way a list is sorted depends on what order is of interest. For example, numbers could be sorted in ascending or descending order, character strings could be sorted lexically or by length, etc.. The value of p is therefore needed in sorting applications to specify the order. It contains the virtual code for a partial order relational operator. This operator can be evaluated with any pair of items selected from a list, and should have a value of true if the left one should precede the right under the ordering. For example, if numbers were to be sorted in ascending order, then p would compute the "less or equal" relation, returning true if its operand were a pair of numbers in which the left is less or equal to the right.

The virtual code semantics for sorting applications is given by these two properties, wherein p is a non-`nil` tree, and `insert` is a `silly` mnemonic to be defined next.

P31 $([[\texttt{sort}]]\ p)\ \texttt{nil} = \texttt{nil}$

P32 $([[\texttt{sort}]]\ p)\ \texttt{(}x\texttt{,}y\texttt{)} = ([[\texttt{insert}]]\ p)\ \texttt{(}x\texttt{,}([[\texttt{sort}]]\ p)\ y\texttt{)}$

These properties say that the empty list is already sorted, and a non-empty list is sorted if its tail is sorted and the head is then inserted in the proper place. This specification of sorting has nothing to do with the sorting algorithm that `avram` really uses.

The meaning of insertion is convenient to specify in virtual code itself. It should satisfy these three equations.

E4 $([[\texttt{insert}]]\ p)\ \texttt{(}x\texttt{,nil)} = \texttt{(}x\texttt{,nil)}$

E5 $([[\texttt{insert}]]\ p)\ \texttt{(}x\texttt{,(}y\texttt{,}z\texttt{))} = \texttt{(}y\texttt{,}([[\texttt{insert}]]\ p)\ \texttt{(}x\texttt{,}z\texttt{))}$ if $p\texttt{(}x\texttt{,}y\texttt{)} = \texttt{nil}$

E6 $([[\texttt{insert}]]\ p)\ \texttt{(}x\texttt{,(}y\texttt{,}z\texttt{))} = \texttt{(}x\texttt{,(}y\texttt{,}z\texttt{))}$ if $p\texttt{(}x\texttt{,}y\texttt{)}$ is a non-`nil` tree

Intuitively, the right argument, whether `nil` or $\texttt{(}y\texttt{,}z\texttt{)}$, represents a list that is already sorted. The left argument x therefore only needs to be compared to the head element, y to ascertain whether or not it belongs at the beginning. If not, it should be inserted into the tail.

A possible implementation of `insert` in `silly` is given in Section B.2 [Insert], page 130.

2.7.13.5 Transfer

A particular interpretation is given to virtual code in the following form.

T26 $[[\texttt{transfer}]]\ f = \texttt{((nil,nil),(nil,(nil,}f\texttt{)))}$

When code in this form is evaluated with an argument, the tree f is used as a state transition function, and the argument is used as a list to be traversed. The traversal begins with f being evaluated on `nil` to get the initial state and the initial output. Thereafter, each item of the list is paired with the current state to be evaluated with f, resulting in a list of output and the next state. The output resulting from the entire application is the cumulative concatenation of all outputs obtained in the course of evaluating f. The computation terminates when f yields a `nil` result. If the list of inputs runs out before the computation terminates, `nil` values are used as inputs.

This behavior is specified more precisely in the following property of the operator, which applies in the case of non-`nil` f.

P33 $([\![\texttt{transfer}]\!]\ f)\ x = ([\![\texttt{transition}]\!]\ f)$ `(nil,(`f `nil,`x`))`

Much of the `transfer` semantics is implicit in the meaning of `transition`. For any given application f, $[\![\texttt{transition}]\!]\ f$ is the virtual code for a function that takes the list traversal from one configuration to the next. A configuration is represented as a tuple, usually in the form `(`*previous outputs*`,((`*state*`,`*output*`),(`*next input*`,`*subsequent inputs*`)))`. A terminal configuration has the form `(`*previous outputs*`,(nil,(`*next input*`,`*subsequent inputs*`)))`. A configuration may also have `nil` in place of the pair `(`*next input*`,`*subsequent inputs*`)` if no more input remains.

In the non-degenerate case, the meaning of $[\![\texttt{transition}]\!]\ f$ is consistent with the following equation.

E7 $([\![\texttt{transition}]\!]\ f)\ \texttt{(}y\texttt{,((}s\texttt{,}o\texttt{),(}i\texttt{,}x\texttt{)))} =$
$([\![\texttt{transition}]\!]\ f)\ \texttt{((}o\texttt{,}y\texttt{),(}f\ \texttt{(}s\texttt{,}i\texttt{),}x\texttt{))}$

That is, the current output o is stored with previous outputs y, the next input i is removed from the configuration, and the next state and output are obtained from the evaluation of f with the state s and the next input i.

In the case where no input remains, the transition function is consistent with the following equation.

E8 $([\![\texttt{transition}]\!]\ f)\ \texttt{(}y\texttt{,((}s\texttt{,}o\texttt{),nil))} =$
$([\![\texttt{transition}]\!]\ f)\ \texttt{((}o\texttt{,}y\texttt{),(}f\ \texttt{(}s\texttt{,nil),nil))}$

This case is similar to the previous one except that the `nil` value is used in place of the next input. Note that in either case, nothing about f depends on the particular way configurations are represented, except that it should have a state as its left argument and an input as its right argument.

Finally, in the case of a terminal configuration, the transition function returns the cumulative output.

E9 $([\![\texttt{transition}]\!]\ f)\ \texttt{(}y\texttt{,(nil,}x\texttt{))} = [\![\texttt{reduce(cat,nil)}]\!]\ [\![\texttt{reverse}]\!]\ y$

The `silly` code `reduce(cat,nil)` has the effect of flattening a list of lists into one long list, which is necessary insofar as the transition function will have generated not necessarily a single output but a list of outputs on each iteration. The `cat` mnemonic stands for list concatenation and is explained in Section 2.7.14.1 [Cat], page 52. The reversal is necessary to cause the first generated output to be at the head of the list. List reversal is a built in operation of the virtual machine and is described in Section 2.7.14.2 [Reverse], page 52.

If such a function as `transition` seems implausible, its implementation in `silly` can be found in Section B.4 [Transition], page 132.

It is usually more awkward to express a function in terms of a `transfer` than to code it directly using recursion or other list operations. However, this feature is provided by the virtual machine for several reasons.

- Functions in this form may be an easier translation target if the source is an imperative language.
- Translating from virtual code to asynchronous circuits or process networks has been a research interest of the author, and code in this form lends itself to easy recognition and mapping onto discrete components.

- The '`--byte-transducer`' and '`--interactive`' command line options to `avram` cause an application to be invoked in a similar manner to the transition function in a `transfer` function, so this feature allows for easy simulation and troubleshooting of these applications without actually deploying them.

2.7.13.6 Mapcur

An alternative form of recursive definition is the following.

T27 [[`mapcur`]] p = `((nil,nil),((nil,nil),(`p`,nil)))`

This form is convenient for applications that cause themselves to be applied recursively to a list of arguments. It has this semantics.

P34 ([[`mapcur`]] p) x = [[`map meta`]] [[`distribute`]] ([[`field`]] p) x

2.7.14 List Functions

In addition to the foregoing list operations, the virtual machine provides a number of canned functions operating on lists, namely concatenation, reversal, distribution, and transposition. These functions could be coded by other means if they were not built in, but the built in versions are faster and smaller.

2.7.14.1 Cat

The list concatenation operation has this representation in virtual code.

T28 [[`cat`]] = `((nil,nil),(nil,nil))`

This function takes a pair of lists as an argument, an returns the list obtained by appending the right one to the left. The semantics of concatenation is what one would expect.

P35 [[`cat`]] `(nil,`z`)` = z

P36 [[`cat`]] `((`x`,`y`),`z`)` = `(`x`,`[[`cat`]] `(`y`,z))`

2.7.14.2 Reverse

The function that reverses a list has the following representation in virtual code.

T29 [[`reverse`]] = `((nil,nil),(nil,(nil,nil)))`

This function takes a list as an argument, and returns a the list consisting of the same items in the reverse order. The semantics is given by the following properties.

P37 [[`reverse`]] `nil` = `nil`

P38 [[`reverse`]] `(`x`,`y`)` = [[`cat`]] `(`[[`reverse`]] y`,(`x`,nil))`

2.7.14.3 Distribute

The function with the following virtual code representation is frequently useful for manipulating lists.

T30 `distribute` = `(((nil,nil),nil),nil)`

This function takes a pair whose right side represents a list, and returns a list of pairs, with one pair for each item in the list. The left side of each pair is the left side of the original argument, and the right side is the corresponding item of the list. A semantics for this operation is specified by the following properties.

P39 [[`distribute`]] `(`x`,nil)` = `nil`

P40 [[`distribute`]] `(`x`,(`y`,`z`))` = `((`x`,`y`),`[[`distribute`]] `(`x`,`z`))`

2.7.14.4 Transpose

The `transpose` operation has the following representation in virtual code.

T31 [[`transpose`]] = `((nil,nil),((nil,nil),(nil,nil)))`

This function takes a list of equal length lists as an argument, and returns a list of lists as a result. In the resulting list, the first item is the list of all first items of lists in the argument. The next item is the list of all second items, and so on.

In the specification of the semantics, the `silly` mnemonic `flat` is defined by `flat = reduce(cat,nil)` in the standard `silly` prelude, which means that it flattens a list of lists into one long list.

P41 [[`transpose`]] x = `nil` if [[`flat`]] x = `nil`

P42 [[`transpose`]] x = `(`[[`map left`]] x`,`[[`transpose`]] [[`map right`]] x`)`
if [[`flat`]] x is a non-`nil` tree

2.7.15 Exception Handling

In quite a few cases, the properties given for the operator up to this point do not imply any particular result. A good example would be an expression such as [[`left`]] `nil`, which appears to represent the left side of an empty pair. It can be argued that expressions like this have no sensible interpretation and should never be used, so it would be appropriate to leave them undefined. On the other hand, attempts to evaluate such expressions occur frequently by mistake, and in any case, the virtual machine emulator should be designed to do something reasonable about them if only for the sake of reporting the error. The chosen remedy for this situation addresses the need for error reporting, and also turns out to be useful in other ways.

2.7.15.1 A Hierarchy of Sets

As indicated already, the virtual machine represents all functions and data as members of a set satisfying the properties in Section 2.1 [Raw Material], page 19, namely a `nil` element and a `cons` operator for constructing trees or nested pairs of `nil`. However, it will be necessary to distinguish the results of computations that go wrong for exceptional reasons from normal results. Because any tree in the set could conceivably represent a normal result, we need to go outside the set to find an unambiguous representation of exceptional results.

Because there may be many possible exceptional conditions, it will be helpful to have a large set of possible ways to encode them, and in fact there is no need to refrain from choosing a countably infinite set. Furthermore, it will be useful to distinguish between different levels of severity among exceptional conditions, so for this purpose a countably infinite hierarchy of mutually disjoint sets is used.

In order to build on the theory already developed, the set that has been used up to this point will form the bottom level of the hierarchy, and its members will represent normal computational results. The members of sets on the higher levels in the hierarchy represent exceptional results. To avoid ambiguity, the term "trees" is reserved for members of the bottom set, as in "for any tree x . . .". Unless otherwise stated, variables like x and y are universally quantified over the bottom set only.

Because each set in the hierarchy is countably infinite, it is isomorphic to the bottom set. With respect to an arbitrary but fixed bijection between them, let `x_n` denote the image in the nth level set of a tree x in the bottom set. The level numbers in this notation start with zero, and we take `x_0` to be synonymous with x. For good measure, let `(x_n)_m` = `x_(n+m)`.

2.7.15.2 Operator Generalization

Each set in the hierarchy induces a structure preserving `cons` operator, denoted `cons_n` for the nth level set, and satisfying this equation.

E10 `cons_n(x_n,y_n)` = `(cons(x,y))_n`

It will be convenient to generalize all of these `cons` operators to be defined on members of other sets than their own.

E11 For m greater than n, `cons_n(x_m,y_p)` = `x_m`

In this equation, p is unrestricted. The intuition is that if the left operand of a `cons` is the result of a computation that went wrong due to an exceptional condition (more exceptional than n, the level already in effect), then the exceptional result becomes the whole result.

It is tempting to hazard a slightly stronger statement, which is that this equation holds even if `y_p` is equal to some expression f x that is undefined according to the operator semantics. This stipulation would correspond to an implementation in which the right operand isn't evaluated after an error is detected in the left, but there are two problems with it.

- This semantics might unreasonably complicate a concurrent implementation of the virtual machine. If evaluation leads to non-termination in some cases where the result is undefined (as it certainly would in any possible implementation consistent with cases where it's defined), then the mechanism that evaluates the right side of a pair must be interruptible in case an exception is detected in the left.
- It is beyond the expressive power of the present mathematical framework to make such a statement, because it entails universal quantification over non-members of the constructed sets, which includes almost everything.

Nevertheless, the implementation in `avram` is sequential and does indeed behave as proposed, with no practical difficulty. As for any deficiency in the theory, it could be banished by recasting the semantics in terms of a calculus of expressions with formal rules of manipulation. An operand to the `cons` operator would be identified not with a member of a semantic domain, but with the expression used to write it down, and then even "undefinedness" could be defined. However, the present author's preference in computing as in life is to let some things remain a mystery rather than to abandon the quest for meaning entirely.

A comparable condition applies in cases where the right side of a pair represents an exceptional result.

E12 For m greater than n, `cons_`n`(`x`_`n`,`y`_`m`)` $=$ y`_`m

Whereas an infinitude of `cons` operators has been needed, it will be possible to get by with only one invisible operator, as before, by generalizing it in the following way to operands on any level of the hierarchy.

P43 f`_`n x`_`n $=$ `(`f x`)_`n

P44 For distinct n and m, f`_`n x`_`m $=$ x`_`m

That is, the result of evaluating two operands on the same level is the image relative to that level of the result of their respective images on the bottom level, but the result of evaluating two operands on different levels is the same as the right operand.

2.7.15.3 Error Messages

The basic strategy for representing the results of exceptional conditions arising from the evaluation of operands on a given level of the hierarchy will be to use an error message corresponding to the image of a list of character strings on the level above.

Unfortunately, the official `silly` standard does not define character constants, but they are available as a vendor specific extension in `silly-me` (millennium edition), where character strings may be enclosed in single quotes. The value of the semantic function [[. . .]] in the case of a character string is the list of representations of the characters, based on Appendix A [Character Table], page 123 and Section 2.4 [Representation of Numeric and Textual Data], page 23.

For the sake of consistency, each standard error message is a list of character strings, even though the list has only one string in it. If any exceptional condition is the result of a computation, it is written to standard error by `avram` as the list of character strings it represents.

P45 ([[`compare`]] `nil)_`n $=$ [[`('invalid comparison',nil)`]]`_(`n`+1)`

P46 ([[`left`]] `nil)_`n $=$ [[`('invalid deconstruction',nil)`]]`_(`n`+1)`

P47 ([[`right`]] `nil)_`n $=$ [[`('invalid deconstruction',nil)`]]`_(`n`+1)`

P48 (([[`fan`]] f) `nil)_`n $=$ [[`('invalid deconstruction',nil)`]]`_(`n`+1)`

P49 ([[`member`]] `nil)_`n $=$ [[`('invalid membership',nil)`]]`_(`n`+1)`

P50 ([[`distribute`]] `nil)_`n $=$ [[`('invalid distribution',nil)`]]`_(`n`+1)`

P51 ([[`cat`]] `nil)_`n $=$ [[`('invalid concatenation',nil)`]]`_(`n`+1)`

P52 ([[`meta`]] `nil`)_n = [[`('invalid recursion',nil)`]]_(n+1)

Note that by virtue of property *P44*, there is no need for an application to make explicit checks for exceptional results at any point, because the exceptional result propagates through to the output of any function composed with the one that incurred it. For example, an application of the form `h = compose(f,right)`, which will cause an invalid deconstruction error if applied in filter mode to an empty file, imposes no requirement that `f` be written to accommodate that possibility (i.e., by checking for it) in order for the error to be reported properly. The following proof demonstrates that the meaning of `f` is irrelevant to the result.

[[`compose(f,right)`]]_0 `nil_0`

= [[`f`]]_0 [[`right`]]_0 `nil_0`

= [[`f`]]_0 [[`('invalid deconstruction',nil)`]]_1

= [[`('invalid deconstruction',nil)`]]_1

In an application `h = compose(f,g)`, the input validation therefore may be confined to the "front end", `g`.

It will be recalled from the discussions of `recur` (Section 2.7.9.1 [Recur], page 44) and `transpose` (Section 2.7.14.4 [Transpose], page 53) that the semantics of virtual code involving these forms is defined in terms of the `field` format for deconstruction functions (Section 2.7.8.1 [Field], page 42), which depends implicitly on the semantics of `left` and `right`, being a generalization of them. An invalid deconstruction message could therefore result from applications incorporating any of the forms of `recur`, `transpose`, or `field`. Invalid deconstructions could also arise from the `replace` operation (Section B.3 [Replace], page 130), which is used for assignment (Section 2.7.10 [Assignment], page 45), because `replace` is defined by virtual code, except as noted next.

2.7.15.4 Expedient Error Messages

Because there are so many ways to cause an invalid deconstruction, this message is the most common in practice and therefore the least informative. As a matter of convenience, `avram` takes the liberty of a slight departure from the virtual machine specification as written hitherto, and employs the following messages when invalid deconstructions occur respectively in the cases of recursion, transposition, and assignment.

- `invalid recursion`
- `invalid transpose`
- `invalid assignment`

That is, this section contradicts and supersedes what is stated at the end of Section 2.7.15.3 [Error Messages], page 55 and implied by the operator properties *P14*, *P16*, and *P42*. It is also possible that user applications may modify the error messages by methods described in Section 2.7.15.5 [Computable Error Messages], page 57.

Whereas these three cases constitute an expedient variation on the semantics, there is another sense in which no possible implementation could conform faithfully to the specification. When an evaluation can not be carried out because of insufficient space on the host machine, one of the following error messages may be the result.

- `memory overflow`
- `counter overflow`

These messages are treated in the same way as those that are caused by programming errors, and propagate to the final result written to standard error without any specific consideration by the application developer. The latter occurs only in connection with the built in weight function (Section 2.7.7.4 [Weight], page 42). Other messages listed in Section 1.6.5 [Application Programming Errors], page 12 are also of this ilk.

2.7.15.5 Computable Error Messages

The automatic generation and reporting of error messages provides a reasonable default behavior for applications that do not consider exceptional conditions. All applications and their input data are ordinarily members of the bottom level set in the hierarchy (Section 2.7.15.1 [A Hierarchy of Sets], page 53). The error messages caused by invalid operations on this level are on the first level above the bottom, which are recognized as such and written to standard error without intervention from the application. However, there are two drawbacks to this style of dealing with exceptions.

- An application developer may wish to translate error messages into terms that are meaningful to the user, not only by literally translating them from English to the local vernacular, but perhaps by relating the particular exceptional condition to application specific causes. While it is convenient for the "back end" code not to be required to intervene in the error reporting, it would be most inconvenient for it not to be able to do so.
- Some application specific errors might not correspond directly to any of the particular conditions detected automatically due to invalid operations, for example a semantic error in a syntactically correct input file. It might be convenient in such cases for an application to be able to define its own error messages but still have them reported automatically like the built in messages.

These situations suggest a need for some ability on the part of an application to operate on error messages themselves. Based on the operator semantics given so far, such an application is inexpressible, because for any application *f*`_0` and error message *x*`_1`, property *P44* stipulates *f*`_0` *x*`_1` = *x*`_1`, meaning that the resulting error message is unchanged. Therefore, we need to define another basic property of the operator.

The following form of virtual code is used in applications that may need to operate on error messages.

T32 [[`guard`]] `(`*f*`,`*g*`)` = `((nil,`*f*`),`*g*`)`

Code in this form has the following semantics.

P53 `([[guard]] (`*f*`,`*g*`))_`*n* *x*`_`*p* = *g*`_(`*n*`+1)` *f*`_`*n* *x*`_`*p*

The intuitive explanation is that *f* is the main part of the application, and *g* is the part of the application that operates on the error message that comes from *f* if an exception occurs while it is being evaluated (i.e., the "exception handler"). Typically the exception handler code implements a function that takes an error message as an argument and returns an error message as a result.

Where there is no exception, the exception handler `g_(n+1)` is never used, because its argument will be on level n, and therefore unaffected by an application on level `n+1`.

Exception handlers may have their own exception handlers, which will be invoked if the evaluation of the exception handler causes a further exception. Such an exception corresponds semantically to a value on the next level of the hierarchy of sets.

2.7.15.6 Exception Handler Usage

One way for this feature of the virtual machine to be used is to intercept and translate error messages to a more meaningful form. An application guarded as shown below causes messages of invalid deconstruction to be changed to `'syntax error'`.

```
main = guard(
   application,
   conditional(
      bu(compare,('invalid deconstruction',nil)),
      (constant ('syntax error',nil),identity)))
```

The conditional compares its argument to the error message for an invalid deconstruction, and if it matches, the syntax error message is returned, but otherwise the original message is returned. Note that an error message must be in the form of a list of character strings, so that it can be printed. Although the message of `'syntax error'` might not be very informative, at least it looks less like a crash. A real application should of course strive to do better than that.

Exception handling features of the virtual machine can also be adapted by applications to raise their own exceptions with customized messages.

```
error_messenger =
   guard(compose(compare,constant nil),constant ('syntax error',nil))
```

This code fragment implements a function that causes a message of `'syntax error'` to be reported for any possible input. This code works by first causing an invalid comparison and then substituting its own error message. A function that always causes an error is not useful in itself, but might be used as part of an application in the following form.

```
main = conditional(validation,(application,error_messenger))
```

In this case, the application checks the validity of the input with a predicate, and invokes the error messenger if it is invalid.

Although the previous examples return a fixed error message for each possible kind of error, it is also possible to have error messages that depend on the input data, as the next example shows.

```
main = (hired apply)(
   compose(
      bu(guard,some_application),
      (hired constant)(constant 'invalid input was:',identity)),
   identity)
```

If the application causes an exception for any reason, the error message returned will include a complete listing of the input, prefaced by the words `'invalid input was:'`. This particular example works only if the input is a list of character strings, but could be adapted for other types of data by substituting an appropriate formatting function for the first identity.

The formatting function would take the relevant data type to a list of character strings. Another possible variation would be to concatenate the invalid input listing with the error message that was generated, rather than just replacing it.

As the last example may suggest, exception handlers turn out to be an essential debugging tool for functional programs, making them as easy to debug as imperative programs if not more so. This example forms the basis for a higher order function that wraps any given function with an exception handler that prints the argument causing it to crash. For arguments not causing a crash, the behavior is unchanged. Alternatively, code implementing a function that unconditionally reports its argument in an error message can be inserted at a strategic point in the application code similarly to a print statement. Finally, inspired use of exception handlers that concatenate their messages with previously generated messages can show something like a parameter stack dump when a recursively defined function crashes. These are all matters for a language designer and are not pursued further in this document.

2.7.16 Interfaces to External Code

A few other combinators have been incorporated into the virtual machine as alternatives to the style of interactive applications described in Section 2.6.4 [Output From Interactive Applications], page 30. These make it possible to interface with external libraries and applications either by a simple function call, or by executing a run-time generated transducer as described previously. In either case, there is no need for any particular command line options to specify interactive invocation, nor for the application to be designed that way from the outset. Existing virtual code applications may therefore be enhanced to make use of these features without radical changes.

To account for these additional capabilities, it is not entirely adequate to continue defining the virtual machine semantics in terms of a mathematical function, but it is done nevertheless due to the lack of any appealing alternative. Although most library functions are in fact functions in the sense that their outputs are determined by their arguments, they defy a concise specification within the present mathematical framework, especially insofar as they may involve finite precision floating point numbers. More problematically, the effect of interaction with a shell is neither well defined nor deterministic. The descriptions that follow presuppose a computational procedure associated with the following definitions but leave its exact nature unspecified.

2.7.16.1 Library combinator

The simplest and fastest method of interfacing to an external library is by way of a virtual machine combinator called `library`. It takes two non-empty character strings as arguments to a virtual code program of the form implied by the following property.

T33 $[[$`library`$]]\ (x,y) =$ `((nil,nil),((`x`,`y`),(nil,nil)))`

Intuitively, x is the name of a library and y is the name of a function within the library. For example, if x is `'math'` and y is `'sqrt'`, then `library`(x,y) represents the function that computes the square root of a floating point number as defined by the host machine's native C implementation, normally in IEEE double precision format. Different functions and libraries may involve other argument and result types, such as complex numbers, arrays, sparse matrices, or arbitrary precision numbers. A list of currently supported external library names with their functions and calling conventions is given in Appendix D [External Libraries], page 135.

On the virtual code side, all function arguments and results regardless of their types are encoded as nested pairs of `nil`, as always, and may be manipulated or stored as any other data, including storage and retrieval from files in '`.avm`' virtual code format (Section 2.3 [File Format], page 22). However, on the C side, various memory management and caching techniques are employed to maintain this facade while allowing the libraries to operate on data in their native format. The details are given more fully in the API documentation, particularly in Section 3.1.4 [Type Conversions], page 72 and Section 3.9 [External Library Maintenance], page 110.

While this style is fast and convenient, it is limited either to libraries that have already been built into the virtual machine, or to those for which the user is prepared to implement a new interface module in C as described in Section 3.9.2 [Implementing new library functions], page 112.

2.7.16.2 Have combinator

As virtual machine interfaces to external libraries accumulate faster than they can be documented and may vary from one installation to another, it is helpful to have a way of interrogating the virtual machine for an up to date list of the installed libraries and functions. A combinator called `have` can be used to test for the availability of a library function. It takes the form

T34 [[`have`]] (x,y) = `((nil,nil),((nil,`x`),(nil,`y`)))`

where x is the name of a library and y is the name of a function within the library encoded as character strings. For example, if x is `'mtwist'` and y is `'u_disc'` (for the natural random number generator function in the Mersenne twistor library) then `have(`x`,`y`)` is a function that returns a non-empty value if an only if that library is installed and that function is available within it. The actual argument to the function is ignored as the result depends only on the installed virtual machine configuration. In this sense, it acts like a `constant` combinator.

One way for this combinator to be used is in code of the form

```
portable_rng =

conditional(
   have('mtwist','u_disc'),
   library('mtwist','u_disc'),
   some_replacement_function)
```

which will use the library function if available but otherwise use a replacement function. Code in this form makes the decision at run time, but it is also possible to express the function such that the check for library presence is made at compile time, as the following example shows, which will imply a slight improvement in performance.

```
non_portable_rng =

apply(
   conditional(
```

```
          have('mtwist','u_disc'),
          constant library('mtwist','u_disc'),
          constant some_replacement_function),
       0)
```

This program would be non-portable in the sense that it would need to be recompiled for each installation if there were a chance that some of them might have the `mtwist` library and some might not, whereas the previous example would be binary compatible across all of them.[2]

The actual value returned by a function `have(foo,bar)` is the list of pairs of strings `<(foo,bar)>` if the function is available, or the empty list otherwise. A non-empty list is represented as a pair `(head,tail)`, and an empty list as `nil`. The angle bracket notation `<a,b,c...>` used here is an abbreviation for `(a,(b,(c...nil)))`.

Either or both arguments to the `have` combinator can be a wildcard, which is the string containing a single asterisk, `'*'`. In that case, the list of all available matching library names and function names will be returned. This feature can be used to find out what library functions are available without already knowing their names.

If a library had a function named `'*'`, which clashes with the wild card string, the interpretation as a wild card would take precedence.

2.7.16.3 Interaction combinator

A further combinator allows virtual code applications to interact directly with any interactive console application using the `expect` library. The mechanism is similar to that of interactive applications documented in the Section 2.6.4 [Output From Interactive Applications], page 30, but attempts to be more convenient. Instead of being designed as an interactive application, any virtual code application may use this combinator to spawn a shell and interact with it in order to compute some desired result.

The advantage of this combinator over the `library` combinator is that it requires no modification of the virtual machine to support new applications. It can also interact with applications that may reside on remote servers, that are implemented languages other than C, or whose source code is unavailable. For example, the GNU R statistical package provides an interactive command to evaluate multivariate normal distribution functions with an arbitrary covariance matrix, but the corresponding function is not provided by the `Rmath` C library (or any other free library, to the author's knowledge) because it is implemented in interpreted code. This combinator makes it callable by an `avram` virtual code application nevertheless. The disadvantage compared to the `library` combinator is that there is more overhead in spawning a process than simply making a call to a built in function, and the programming interface is more complicated.

The combinator takes the form

T35 $[[\texttt{interact}]]\ f = \texttt{((nil,nil),(((nil,nil),nil),((nil,}f\texttt{),nil)))}$

where f is the virtual code for a function that follows the same protocol described in Section 2.6.4 [Output From Interactive Applications], page 30, except that it does not allow

[2] In practice both examples are equally portable because the `mtwist` source is distributed with `avram` so all installations will have it. Most libraries are distributed separately.

file output as described in Section 2.6.4.3 [Mixed Modes of Interaction], page 33. The argument `x` is ignored when the expression `(interact f) x` is evaluated, similarly to the way the argument is ignored in an expression like `(constant k) x`. The result returned is a transcript of the dialogue that took place between `f` and the externally spawned shell, represented as a list of lists of strings for line oriented interaction, or a list of characters alternating with lists of strings in the case of character oriented interaction.

The following example demonstrates a trivial use of the `interact` combinator to spawn an `ftp` client, do an `ls` command, and then terminate the session.

```
eof = <(nil,(nil,(((nil,nil),nil),(nil,nil))))>

demo =

interact conditional(
   conditional(identity,constant false,constant true),
   constant(0,<'ftp'>,<'ftp> '>),
   conditional(
      conditional(left,constant false,constant true),
      constant(1,<'ls','>,<'','ftp> '>),
      conditional(
         compose(compare,couple(left,constant 1)),
         constant(2,<'bye',''>,<eof>),
         constant nil)))
```

Some liberties are taken with `silly` syntax in this example, in the way of using angle brackets to denote lists, and numbers to represent states.

- The interacting transducer works by checking whether its argument is empty (via the `identity` function used as a predicate in the `conditional`, which is then negated). In that case it returns the triple containing the initial state of 0, the `ftp` shell command to spawn the client, and the `'ftp> '` prompt expected when the client has been spawned, both of the latter being lists of strings.
- If the argument is non-empty, then next it checks whether it is in the initial state of 0, (via the `left` function used as a predicate, referring to the state variable expected on the left of any given `(state,input)` pair, also negated). If so, it returns the triple containing the next state of 1, the `ls` command followed by an empty string to indicate a line break, and the expected prompt preceded by an empty string to match it only at the beginning of a line.
- Finally, it checks for state 1, in which case it issues the `bye` command to close the session, `eof` rather than a prompt to wait for termination of the client, and a state of 2.
- In the remaining state of 2, which needn't be explicitly tested because it is the only remaining possibility, the program returns a `nil` value to indicate that the computation has terminated.

Deadlock would be possible at any point if either party did not follow this protocol, but for this example it is not an issue. If an expression of the form `demo x` were to be evaluated, then regardless of the value of `x`, the value of the result would be as shown below.

```
<
   <'ftp'>,
   <'ftp> '>,
   <'ls',''>,
   <'ls','Not connected.','ftp> '>,
   <'bye',''>,
   <'bye',''>>
```

That is, it would be a list of lists of strings, alternating between the output of the interactor and the output of the `ftp` client. If the spawned application had been something non-trivial such as a computer algebra system or a command line web search utility, then it is easy to see how functions using this combinator can leverage off a wealth of available resources.

2.7.17 Vacant Address Space

Not every possible pattern has been used by the virtual machine as a way of encoding a function. The following patterns, where a, b, and c are non-`nil` trees, do not represent anything useful.

unary forms

```
((nil,nil),((nil,nil),(nil,((nil,a),nil))))
((nil,nil),((nil,nil),(nil,(nil,(nil,a)))))
```

binary forms

```
((nil,nil),((nil,nil),(a,b)))
((nil,nil),((a,nil),(b,nil)))
((nil,nil),((a,nil),(nil,b)))
```

ternary forms

```
((nil,nil),((a,b),(c,nil)))
((nil,nil),((a,b),(nil,c)))
((nil,nil),((a,nil),(b,c)))
((nil,nil),((nil,a),(b,c)))
```

These patterns are detected by the virtual machine simply to avoid blowing it up, but they always cause an error message to be reported.

P55 For f matching any of the first three trees in the above list,
$f\_n\ x\_n = [[$`('unsupported hook',nil)`$]]\_(n+1)$

P56 For the remaining trees f in the above list,
$f\_n\ x\_n = [[$`('unrecognized combinator (code m)',nil)`$]]\_(n+1)$

Here, m is a numeric constant dependent on which tree f was used. The unsupported hook message is meant to be more informative than the unrecognized combinator message, suggesting that a feature intended for future use is not yet available.

This list has been assembled for the benefit of readers considering the addition of backward compatible extensions to the virtual code semantics, who are undeterred by the facts that

- the computational model is already universal
- virtual code applications are already interoperable with all kinds of high performance software having a text based or console interface by way of the `interact` combinator

- an unlimited number of built in library functions can be added by way of the `library` combinator as described in Section 3.9.2 [Implementing new library functions], page 112
- the C code in `avram` makes fairly intricate use of pointers with a careful policy of reference counting and storage reclamation
- there is also a performance penalty incurred by further extensions to the semantics, even for applications that don't use them, because a pattern recognition algorithm in the interpreter has more cases to consider.

Nevertheless, a new functional form combining a pair of functions to be interpreted in a new way by the virtual machine could be defined using any of the binary forms above, for example, with a as the virtual code for one of the functions and b as that of the other. Such a form would not conflict with any existing applications, provided that both a and b are not `nil`, which is true of any valid representation for a function.

Virtual machine architects, take note. There are infinitely many trees fitting these patterns, but it would be possible to use them up by assigning them without adequate foresight. For example, if interpretations were assigned to the four ternary forms, the three binary forms, and one of the remaining unary forms, then the only unassigned pattern could be of the form

`((nil,nil),((nil,nil),(nil,(nil,(nil,`a`)))))`

Assigning an interpretation to it would leave no further room for backward compatible expansion. On the other hand, any tree of the following form also fits the above pattern,

`((nil,nil),((nil,nil),(nil,(nil,(nil,(`b`,`c`))))))`

with any values for b and c. Different meanings could be chosen for the case where both are `nil`, both are non-`nil`, or one is `nil` and the other non-`nil`, allowing two unary forms, one binary, and one constant. If at least one of these patterns is reserved for future enhancements, then a potentially inexhaustible supply of address space remains and there will be no need for incompatible changes later.

3 Library Reference

Much of the code developed for `avram` may be reusable in other projects, so it has been packaged into a library and documented in this chapter. For ease of reference, this chapter is organized with a separate section for each source file. For the most part, each source file encapsulates an abstract type and a number of related functions, except for a few cases where C makes such a design awkward. An attempt has been made to present the sections in a readable order as far as possible.

The documentation in this chapter is confined to the application program interface (API), and does not delve unnecessarily into any details of the implementation. A reader wishing to extend, modify, or troubleshoot the library itself can find additional information in the source code comments. These are more likely to be in sync with the code than this document may be, and are more readily accessible to someone working with the code.

Some general points pertaining to the library are the following.

- Unlike the previous chapter, this chapter uses the word "function" in the C sense rather than the mathematical sense of the word.
- Internal errors are internal from the user's point of view, not the developer's (Section 1.6.1 [Internal Errors], page 10). Invoking these functions in ways that are contrary to their specifications can certainly cause internal errors (not to mention segfaults).
- The library is definitely not thread safe, and thread safety is not a planned enhancement. The amount of locking required to make it thread safe would probably incur an objectionable performance penalty due to the complexity of the shared data structures involved, in addition to being very difficult to get right. If you need these facilities in a concurrent application, consider spawning a process for each client of the library so as to keep their address spaces separate.
- The library files are built from the standard source distribution using GNU `libtool`. In the default directory hierarchy, they will be found either in '`/usr/lib/libavram.*`' or in '`/usr/local/lib/libavram.*`'. These directories will differ in a non-standard installation.
- The header files will probably be located in either '`/usr/include/avm/*.h`' or '`/usr/local/include/avm/*.h`' for a standard installation.
- All exported functions, macros and constants are preceded with `avm_`, so as to reduce the chance of name clashes with other libraries. Not all type declarations or field identifiers follow this convention, because that would be far too tedious.
- The library header files are designed to be compatible with C++ but have been tested only with C. Please refer to platform specific documentation for further information on how to link library modules with your own code.

3.1 Lists

The basic data structure used for representing virtual code and data in the `avram` library is declared as a `list`. The `list` type is a pointer to a structure having a `head` field and a `tail` field, which are also lists. The empty tree, `nil`, is represented by the C constant `NULL`. A tree of the form `cons(`a`,`b`)` is represented in C as a list whose `head` is the representation of a and whose `tail` is the representation of b.

A number of other fields in the structure are maintained automatically and should not be touched. For that matter, even the `head` and `tail` fields should be considered read-only. Because of sharing, it is almost never valid to modify a list "in place", except for cases that are already covered by library functions.

3.1.1 Simple Operations

These functions are declared in the header file `lists.h`, which should be included in any C source file that uses them with a directive such as `#include <avm/lists.h>`. All of these functions except the first three have the potential cause a memory overflow. In that event, a brief message is written to standard error and the process is killed rather than returning to the caller. It is possible for client programs requiring more robust behavior to do their own error handling by using the alternative versions of these operations described in the next section.

`void` **avm_initialize_lists** () Function

The function `avm_initialize_lists` should be called before any of the other ones in this section is called, because it sets up some internal data structures. Otherwise, the behavior of the other functions is undefined.

`void` **avm_dispose** (`list` *front*) Function

This function deallocates the memory associated with a given list, either by consigning it to a cache maintained internally by the library, or by the standard `free` function if the cache is full. Shared lists are taken into account and handled properly according to a reference counting scheme. Lists should be freed only by this function, not by using `free` directly.

`void` **avm_count_lists** () Function

If a client program aims to do its own storage reclamation, this function can be called optionally at the end of a run when it is believed that all lists have been freed. If any allocated lists remain at large, a warning will be printed to standard error. This function therefore provides a useful check for memory leaks. Overhead is small enough that it is not infeasible to leave this check in the production code.

`list` **avm_copied** (`list` *operand*) Function

A copy of the argument list is returned by this function. The copy remains intact after the original is reclaimed. A typical use might be for retaining part of a list after the rest of it is no longer needed. In this example, a list `x` is traversed by a hypothetical `visit` function to each item, which is then immediately reclaimed.

```
while(x){
   visit(x->head);
   old_x = x;
   x = avm_copied(x->tail);       /* the right way */
   avm_dispose(old_x);
}
```

This example allows each item in the list to be visited even as previously visited items are reclaimed, because `x` is copied at each iteration. This example contrasts with the next one, which will probably cause a segmentation fault.

```
while(x){
   visit(x->head);
   old_x = x;
   x = x->tail;                  /* the wrong way */
   avm_dispose(old_x);
}
```

In the second example, a reference is made to a part of a list which no longer exists because it has been deallocated.

In fact, the `avm_copied` function does nothing but increment a reference count, so it is a fast, constant time operation that requires no additional memory allocation. Semantically this action is equivalent to creating a fresh copy of the list, because all list operations in the library deal with reference counts properly.

`list` **avm_join** (`list` *left*, `list` *right*) Function

This function takes a pair of lists to a list in which the left is the head and the right is the tail. It may need to use `malloc` to allocate additional memory. If there is insufficient memory, an error message is written to standard error and the program exits. When the list returned by `avm_join` is eventually deallocated, the lists from which it was built are taken with it and must not be referenced again. For example, the following code is an error.

```
z = avm_join(x,y);
...
avm_dispose(z);
avm_print_list(x);         /* error here */
```

To accomplish something similar to this without an error, a copy of `x` should be made, as in the next example.

```
z = avm_join(avm_copied(x),y);
...
avm_dispose(z);
avm_print_list(x);         /* original x still intact */
```

`void` **avm_enqueue** (`list` **front*, `list` **back*, `list` *operand*) Function

A fast simple way of building a list head first is provided by the `enqueue` function. The `front` is a pointer to the beginning of the list being built, and the `back` is a pointer to the last item. The recommended way to use it would be something like this.

```
front = back = NULL;
avm_enqueue(&front,&back,item);
avm_enqueue(&front,&back,next_item);
avm_enqueue(&front,&back,another_item);
...
```

It might be more typical for the calls to `avm_enqueue` to appear within a loop. In any case, after the above code is executed, the following postconditions will hold.

```
front->head == item
front->tail->head == next_item
front->tail->tail->head == another_item
```

```
back->head == another_item
back->tail == NULL
```

The `avm_enqueue` function must never be used on a shared list, because it modifies its arguments in place. The only practical way to guarantee that a list is not shared is to initialize the `front` and `back` to `NULL` as shown before the first call to `avm_enqueue`, and to make no copies of `front` or `back` until after the last call to `avm_enqueue`.

Because a list built with `avm_enqueue` is not shared, it is one of the few instances of a list that can have something harmlessly appended to it in place. For example, if the next line of code were

```
back->tail = rest_of_list;
```

that would be acceptable assuming `rest_of_list` is not shared and does not conceal a dangling or cyclic reference, and if nothing further were enqueued.

The items that are enqueued into a list are not copied and will be deallocated when the list is deallocated, so they must not be referenced thereafter. A non-obvious violation of this convention is implicit in the following code.

```
...
avm_enqueue(&front,&back,x->head);
...
avm_dispose(front);
avm_print_list(x);      /* error here  */
```

This code might cause a segmentation fault because of the reference to `x` after its head has been deallocated. The following code is subject to the same problem,

```
...
avm_enqueue(&front,&back,x->head);
...
avm_dispose(x);
avm_print_list(front);       /* error here */
```

as is the following.

```
...
avm_enqueue(&front,&back,x->head);
...
avm_dispose(x);       /* front is now impossible to reclaim */
avm_dispose(front);
```

The problem with the last example is that it is not valid even to dispose of the same list more than once, albeit indirectly.

If part of a list is intended to be enqueued temporarily or independently of its parent, the list should be copied explicitly, as the following code demonstrates.

```
...
avm_enqueue(&front,&back,avm_copied(x->head));   /* correct */
...
avm_dispose(front);
avm_print_list(x);
```

`counter` **avm_length** (`list` *operand*) Function

A `counter` is meant to be the longest unsigned integer available on the host machine, and is defined in `common.h`, which is automatically included whenever `lists.h` is included. The `avm_length` function returns the number of items in a list. If a list is `NULL`, a value of zero is returned. There is a possibility of a counter overflow error from this function (Section 1.6.3 [Overflow Errors], page 11), but only on a platform where the `counter` type is shorter than the address length.

`counter` **avm_area** (`list` *operand*) Function

This function is similar to `avm_length`, but it treats its argument as a list of lists and returns the summation of their lengths.

`list` **avm_natural** (`counter` *number*) Function

This function takes a `counter` to its representation as a list, as described in Section 2.4 [Representation of Numeric and Textual Data], page 23. That is, the number is represented as a list of bits, least significant bit first, with each zero bit represented by `NULL` and each one bit represented by a list whose `head` and `tail` are `NULL`.

`void` **avm_print_list** (`list` *operand*) Function

The `avm_print_list` function is not used in any production code but retained in the library for debugging purposes. It prints a list to standard output using an expression involving only commas and parentheses, as per the `silly` syntax (Section 2.7.4 [A Simple Lisp Like Language], page 35). The results quickly become unintelligible for lists of any significant size. The function is recursively defined and will crash in the event of a stack overflow, which will occur in the case of very large or cyclic lists.

`list` **avm_position** (`list` *key*, `list` *table*, `int` **fault*) Function

This function searches for a *key* in a short *table* where each item is a possible key.

If it's not found, a `NULL` value is returned. If it's found, a list representing a character encoding according to Appendix A [Character Table], page 123 is returned.

The ascii code of the character corresponding to the returned list is the position of the *key* in the *table*, assuming position numbers start with 1.

The table should have a length of 255 or less. If it's longer and the *key* is found beyond that range, the higher order bits of the position number are ignored.

The integer referenced by *fault* is set to a non-zero value in the event of a memory overflow, which could happen in the course of the list comparisons necessary for the search.

3.1.2 Recoverable Operations

The functions in this section are similar to the ones in the previous section except with regard to error handling. Whereas the other ones cause an error message to be printed and the process to exit in the event of an overflow, these return to the caller, whose responsibility it is to take appropriate action. The functions in both sections are declared in '`lists.h`', and should be preceded by a call to `avm_initialize_lists`.

`list` **avm_recoverable_join** (`list` *left*, `list` *right*) Function

This function is similar to `avm_join`, but will return a `NULL` pointer if memory that was needed can not be allocated. A `NULL` pointer would never be the result of a join under normal circumstances, so the overflow can be detected by the caller. Regardless of whether overflow occurs, the arguments are deallocated by this function and should not be referenced thereafter.

`void` **avm_recoverable_enqueue** (`list` `*`*front*, `list` `*`*back*, `list` *operand*, `int` `*`*fault*) Function

This version of the enqueue function will dispose of the *operand* if there isn't room to append another item and set `*`*fault* to a non-zero value. Other than that, it does the same as `avm_enqueue`.

`counter` **avm_recoverable_length** (`list` *operand*) Function

This function checks for arithmetic overflow when calculating the length of a list, and returns a zero value if overflow occurs. The caller can detect the error by noting that zero is not the length of any list other than `NULL`. This kind of overflow is impossible unless the host does not have long enough integers for its address space.

`counter` **avm_recoverable_area** (`list` *operand*, `int` `*`*fault*) Function

This function is similar to `avm_area`, except that it reacts differently to arithmetic overflow. The `fault` parameter should be the address of an integer known to the caller, which will be set to a non-zero value if overflow occurs. In that event, the value of zero will also be returned for the area. Note that it is possible for non-empty lists to have an area of zero, so this condition alone is not indicative of an error.

`list` **avm_recoverable_natural** (`counter` *number*) Function

This function returns the `list` representation of a native unsigned long integer, provided that there is enough memory, similarly to the `avm_natural` function. Unlike that function, this one will return a value of `NULL` rather than exiting the program in the event of a memory overflow. The overflow can be detected by the caller insofar as a `NULL` `list` does not represent any number other than zero.

3.1.3 List Transformations

Some functions declared in '`listfuns.h`' are used to implement the operations described in Section 2.7.14 [List Functions], page 52. These functions are able to report error messages in the event of overflow or other exceptional conditions, as described in Section 2.7.15.3 [Error Messages], page 55. The error messages are represented as lists and returned to the caller. The occurrence of an error can be detected by the `*`*fault* flag being set to a non-zero value. None of these functions ever causes a program exit except in the event of an internal error.

`void` **avm_initialize_listfuns** () Function

This has to be called before any of the other functions in this section is called. It initializes the error message lists, among other things.

`void` **avm_count_listfuns** () Function

At the end of a run, a call to this function can verify that no unreclaimed storage attributable to these functions persists. If it does, a warning is printed to standard error. If `avm_count_lists` is also used, it must be called after this function.

`list` **avm_reversal** (`list` *operand*, `int` **fault*) Function

The reversal of the list is returned by this function if no overflow occurs. A non-zero **fault* and an error message are returned otherwise. The original *operand* still exists in its original order after this function is called. The amount of additional storage allocated is proportional only to the length of the list, not the size of its contents.

`list` **avm_distribution** (`list` *operand*, `int` **fault*) Function

This function performs the operation described in Section 2.7.14.3 [Distribute], page 53. The invalid distribution message is returned in the event of a `NULL` operand. Otherwise, the returned value is the distributed list. In any event, the *operand* is unaffected.

`list` **avm_concatenation** (`list` *operand*, `int` **fault*) Function

The *operand* is treated as a pair of lists to be concatenated, with the left one in the `head` field and the right one in the `tail` field. The invalid concatenation message is returned in the event of a `NULL` *operand*. The result returned otherwise is the concatenation of the lists, but the given *operand* still exists unchanged.

`list` **avm_transposition** (`list` *operand*, `int` **fault*) Function

The operation performed by this function corresponds to that of Section 2.7.14.4 [Transpose], page 53. Unlike other functions in this section, the operand passed to this function is deallocated, and must not be referenced thereafter. The transposed list is accessible as the returned value of this function. If the original *operand* is still needed after a call to `avm_transposition`, only a copy of it should be passed to it, obtained from `avm_copied`. The invalid transpose error message is the result if the operand does not represent a list of equal length lists.

`list` **avm_membership** (`list` *operand*, `int` **fault*) Function

This function computes the membership predicate described in Section 2.7.11.2 [Member], page 46. The operand is a list in which the `tail` field is a list that will be searched for the item in the `head`. If the item is not found, a `NULL` list is returned, but otherwise a list with `NULL` `head` and `tail` fields is returned. If the operand is `NULL`, an error message of invalid membership is returned and **fault* is set to a non-zero value.

The `avm_membership` function calls `avm_binary_comparison` in order to compare lists, so the same efficiency and side-effect considerations are relevant to both (Section 3.1.5 [Comparison], page 80). It is not necessary to `#include` the header file `compare.h` or to call `avm_initialize_compare` in order to use `avm_membership`, because they will be done automatically.

`list` **avm_binary_membership** (`list` *operand*, `list` *members*, `int` **fault*); Function

This function is the same as `avm_membership` except that it allows the element and the set of members to be passed as separate lists instead of being the head and the tail of the same list.

`list` **avm_measurement** (`list` *operand*, `int` **fault*) Function

This function implements the operation described in Section 2.7.7.4 [Weight], page 42, which pertains to the weight of a tree. The returned value of this function is a list encoding the weight as a binary number, unless a counter overflow occurs, in which case it's an error message. As noted previously, the weight of a tree can easily be exponentially larger than the amount of memory it occupies, but this function uses native integer arithmetic for performance reasons. Hence, a counter overflow is a real possibility.

3.1.4 Type Conversions

External library functions accessed by the `library` combinator as explained in Section 2.7.16.1 [Library combinator], page 59 may operate on data other than the `list` type usually used by `avram`, such as floating point numbers and arrays, but a virtual code application must be able to represent the arguments and results of these functions in order to use them. As a matter of convention, a data structure occupying *size* bytes of contiguous storage on the host machine appears as a list of length *size* to a virtual code application, in which each item corresponds to a byte, and is represented according to Appendix A [Character Table], page 123.

In principle, a virtual code application invoking a library function to operate on a contiguous block of data, such as an IEEE double precision number, for example, would construct a list of eight character representations (one for each byte in a double precision number), and pass this list as an argument to the library function. The virtual machine would transparently convert this representation to the native floating point format, evaluate the function, and convert the result back to a list. In practice, high level language features beyond the scope of this document would insulate the programmer from some of the details on the application side as well.

To save the time of repeatedly converting between the list representation and the contiguous native binary representation, the structure referenced by a `list` pointer contains a `value` field which is a `void` pointer to a block of memory of unspecified type, and serves as a persistent cache of the value represented by the list. This field normally should be managed by the API rather than being accessed directly by client modules, but see the code in '`mpfr.c`' for an example of a situation in which it's appropriate to break this rule. (Generally these situations involve library functions operating on non-contiguous data.)

3.1.4.1 Primitive types

A pair of functions in support of this abstraction is prototyped in '`listfuns.h`'. These functions will be of interest mainly to developers wishing to implement an interface to a new library module and make it accessible on the virtual side by way of the `library` combinator (Section 2.7.16.1 [Library combinator], page 59).

`void` ***avm_value_of_list** (`list` *operand*, `list` **message*, `int` **fault*) Function

This function takes an *operand* representing a value used by a library function in the format described above (Section 3.1.4 [Type Conversions], page 72) and returns a pointer to the value.

The `value` field in the *operand* normally will point to the block of memory holding the value, and the *operand* itself will be a list of character representations whose binary encodings spell out the value as explained above.

The `value` field need not be initialized on entry but it will be initialized as a side effect of being computed by this function. If it has been initialized due to a previous call with the same *operand*, this function is a fast constant time operation.

The caller should not free the pointer returned by this function because a reference to its value will remain in the *operand*. When the *operand* itself is freed by `avm_dispose` (Section 3.1.1 [Simple Operations], page 66), the value will go with it.

If an error occurs during the evaluation of this function, the integer referenced by *fault* will be set to a non-zero value, and the list referenced by *message* will be assigned a representation of a list of strings describing the error. The *message* is freshly created and should be freed by the caller with `avm_dispose` when no longer needed.

Possible error messages are `<'missing value'>`, in the case of an empty *operand*, `<'invalid value'>` in the case of an *operand* that is not a list of character representations, and `<'memory overflow'>` if there was insufficient space to allocate the result.

`list` **avm_list_of_value** (`void *`*contents*`, size_t` *size*`, int *`*fault*) Function

This function performs the inverse operation of `avm_value_of_list`, taking the address of an area of contiguously stored data and its *size* in bytes to a list representation. The length of the list returned is equal to the number of bytes of data, *size*, and each item of the list is a character representation for the corresponding byte as given by Appendix A [Character Table], page 123.

A copy of the memory area is made so that the original is no longer needed and may be freed by the caller. A pointer to this copy is returned by subsequent calls to `avm_value_of_list` when the result returned by this function is used as the *operand* parameter.

If there is insufficient memory to allocate the result, the integer referenced by *fault* is set to a non-zero value, and a copy of the message `<'memory overflow'>` represented as a list is returned. This function could also cause a segmentation fault if it is passed an invalid pointer or a *size* that overruns the storage area. However, it is acceptable to specify a *size* that is less than the actual size of the given memory area to construct a list representing only the first part of it. The *size* must always be greater than zero.

3.1.4.2 One dimensional arrays

A couple of functions declared in '`matcon.h`' are concerned mainly with one dimensional arrays or vectors. They have been used for vectors of double precision and complex numbers, but are applicable to any base type that is contiguous and of a fixed size.

The motivation for these functions is to enable a developer to present an API to virtual code applications wherein external library functions operating natively on one dimensional arrays of numbers are seen from the virtual side to operate on lists of numbers. Lists are the preferred container for interoperability with virtual code applications.

void ***avm_vector_of_list** (list *operand*, size_t *item_size*, list **message*, int **fault*) — Function

This function calls `avm_value_of_list` (Section 3.1.4.1 [Primitive types], page 72) for each item of the *operand* and puts all the values together into one contiguous block, whose address is returned.

The given *item_size* is required to be the lengths of the items, all necessarily equal, and is required only for validation. For example, *item_size* is 8 for a list of double precision numbers, because they occupy 8 bytes each and are represented as lists of length 8.

The total number of bytes allocated is the product of *item_size* and the length of the *operand*. Unlike the case of `avm_value_of_list` (Section 3.1.4.1 [Primitive types], page 72), the result returned by this function should be explicitly freed by the caller when no longer needed.

Any errors such as insufficient memory cause the integer referenced by *fault* to be assigned a non-zero value and the *message* to be assigned an error message represented as a list of strings. An error message of `<'bad vector specification'>` is possible in the case of an empty *operand* or one whose item lengths don't match the given *item_size*. Error messages caused by `avm_value_of_list` can also be generated by this function. Any non-empty error message should be reclaimed by the caller using `avm_dispose` (Section 3.1.1 [Simple Operations], page 66). If an error occurs, a `NULL` pointer is returned.

list **avm_list_of_vector** (void **vector*, int *num_items*, size_t *item_size*, int **fault*) — Function

This function takes it on faith that an array of dimension *num_items* in which each item occupies *item_size* bytes begins at the address given by *vector*. A list representation of each item in the array is constructed by the function `avm_list_of_value` (Section 3.1.4.1 [Primitive types], page 72), and a list of all of the lists thus obtained in order of their position in the array is returned.

In the event of any errors caused by `avm_list_of_value` or errors due to insufficient memory, the error message is returned as the function result, and the integer referenced by *fault* is assigned a non-zero value. The error message is in the form of a list of character string representations. A segmentation fault is possible if *vector* is not a valid pointer or if the array size implied by misspecified values of *num_items* and *item_size* exceeds its actual size.

3.1.4.3 Two dimensional arrays

Several other functions in 'matcon.h' are meant to support conversions between matrices represented as lists of lists and arrays in a variety of representations. Dense matrices either square or rectangular are accommodated, and symmetric square matrices can be stored with redundant entries omitted in either upper trangular or lower triangular format.

Similarly to the vector operations (Section 3.1.4.2 [One dimensional arrays], page 73) these functions are intended to allow a developer to present an interface to external libraries based on lists rather than arrays.

The preferred convention for virtual code applications is to represent a matrix as a list of lists of entities (typically numbers), with one list for each row of the matrix. For example, a 3 by 3 matrix containing a value of `aij` in the `i`-th row and the `j`-th column would be represented by this list of three lists.

```
<
   <a11,a12,a13>,
   <a21,a22,a23>,
   <a31,a32,a33>>
```

Such a representation is convenient for manipulation by virtual machine combinators, for example `transpose` (Section 2.7.14.4 [Transpose], page 53), and is readily identified with the matrix it represents.

If a matrix is symmetric (that is, with `aij` equal to `aji` for all values of `i` and `j`), only the lower triangular portion needs to be stored because the other entries are redundant. The list representatation would be something like this.

```
<
   <a11>,
   <a21,a22>,
   <a31,a32,a33>>
```

Another alternative for representing a symmetric matrix is to store only the upper triangular portion. In this case, a list such as the following would be used.

```
<
   <a11,a12,a13>,
   <a22,a23>,
   <a33>>
```

The upper and lower triangular representations are distinguishable by whether or not the row lengths form an increasing sequence.

In addition to representing symmetric matrices, these upper and lower triangular forms are also appropriate for representing matrices whose remaining entries are zero, such as the factors in an LU decomposition.

`void` ***avm_matrix_of_list** (`int` *square*, `int` *upper_triangular*, `int` *lower_triangular*, `int` *column_major*, `list` *operand*, `size_t` *item_size*, `list` `*`*message*, `int` `*`*fault*) — Function

This function converts a matrix in one of the list representations above to a contiguous array according to the given specifications. The array can contain elements of any fixed sized type of size *item_size*. The memory for it is allocated by this function and it should be freed by the caller when no longer needed.

The input matrix is given by the list parameter, *operand*, and its format is described by the integer parameters *square*, *upper_triangular*, and *lower_triangular*. The number of bytes occupied by each entry is given by *item_size*.

To the extent these specifications are redundant, they are used for validation. If any of the following conditions is not met, the integer referenced by *fault* is assigned a non-zero value and a copy of the message `<'bad matrix specification'>` represented as a list is assigned to the list referenced by *message*. Errors are also possible due to insufficient memory.

- The *operand* must be a list of lists of lists such that each item of each item is has a length of *item_size*, and its items consist of character representations as required by `avm_value_of_list` (Section 3.1.4.1 [Primitive types], page 72).
- If the lengths of the top level lists in the *operand* form an increasing sequence, the lower triangular representation is assumed and the *lower_triangular* parameter must have a non-zero value.
- If the lengths of the top level lists in the *operand* form a decreasing sequence, the upper triangular representation is assumed and the *upper_triangular* parameter must have a non-zero value.
- At least one of *upper_triangular* or *lower_triangular* must be zero.
- If *square* has a non-zero value, then either all items of the *operand* must have the same length as the operand, or if it's triangular, then the longest one must have the same length as the operand.
- If the *operand* is neither square nor a triangular form, all items of it are required to have the same length.

The parameters *upper_triangular* or *lower_triangular* may be set to non-zero values even if the *operand* is not in one of the upper or lower triangular forms discussed above. In this case, the *operand* must be square or rectangular (i.e., with all items the same length), and the following interpretations apply.

- If *upper_triangular* is non-zero, the diagonal elements and the upper triangular portion of the input matrix are copied to the output. The lower triangle of the input is ignored and the lower triangle of the output is left uninitialized.
- If *lower_triangular* is non-zero, the diagonal elements and the lower triangular portion of the input matrix are copied to the output. The upper triangle of the input is ignored and the upper triangle of the output is left uninitialized.

The *column_major* parameter affects the form of the output array. If it is zero, then each row of the input matrix is stored in a contiguous block of memory in the output array, and if it is non-zero, each column is stored contiguously. The latter representation is also known as Fortran order and may be required by library functions written in Fortran.

In all cases when a triangular form is specified, part of the output matrix is left uninitialized. The redundant entries may be assigned if required by the `avm_reflect_matrix` function (Section 3.1.4.4 [Related utility functions], page 78).

`list` **avm_list_of_matrix** (`void *`*matrix*, `int` *rows*, `int` *cols*, `size_t` *item_size*, `int *`*fault*) Function

This function performs an inverse operation to `avm_matrix_of_list` by taking the address of a matrix stored as a contiguous array in the parameter *matrix* and constructing the list representation as discussed above. Only square and rectangular matrices in row major order are supported, but see `avm_matrix_transposition` for a way to convert between row major and column major order (Section 3.1.4.4 [Related utility functions], page 78).

The parameters *rows*, *cols*, and *item_size* describe the form of the matrix. The list returned as a result will have a length of *rows*, and each item will be a list of length *cols*. Each item of the result corresponds to a row of the matrix, and each item of the items represents the an entry of the matrix as a list of length *item_size*. These items could be passed to `avm_value_of_list`, for example, to obtain their values (Section 3.1.4.1 [Primitive types], page 72).

Memory is allocated by this function to create the list, which can be reclaimed by `avm_dispose` (Section 3.1.1 [Simple Operations], page 66). If there is insufficient memory, the integer referenced by *fault* is assigned a non-zero value and the result returned is a list representation of the message `<'memory overflow'>`. The error message be reclaimed by the caller as well using `avm_dispose`.

A packed storage representation for symmetric square matrices and triangular matrices is of interest because it is used by some library functions, notably those in `LAPACK`, to save memory and thereby accommodate larger problems. In this representation, column major order is assumed, and either the lower or the upper triangle of the matrix is not explicitly stored. For example, a lower triangular matrix whose list representation corresponds to

```
<
   <a11>,
   <a21,a22>,
   <a31,a32,a33>,
   <a41,a42,a43,a44>>
```

would be stored according to the memory map

```
[a11 a21 a31 a41 a22 a32 a42 a33 a43 a44]
```

with `a11` at the beginning address. An upper triangular matrix

```
<
   <a11,a12,a13,a14>,
   <a22,a23,a24>,
   <a33,a34>,
   <a44>>
```

would be stored according to the memory map

```
[a11 a12 a22 a13 a23 a33 a14 a24 a34 a44].
```

A couple of functions converting between list representations and packed array format are provided as described below.

`void` ***avm_packed_matrix_of_list** (`int` *upper_triangular*, `list` *operand*, `int` *n*, `size_t` *item_size*, `list` **message*, `int` **fault*) — Function

If the *operand* is a list in one of the triangular forms explained above, then the *upper_triangular* parameter must be consisitent with it, being non-zero if the *operand* is upper triangular and zero otherwise.

If the *operand* is not in a triangular form, then each item of the operand must be a list of length *n*. In this case, the *upper_triangular* parameter indicates which triangle of the operand should be copied to the result, and the other triangle is ignored.

In either case, the operand must have a length of *n*, and the items of its items must be lists of length *item_size* containing character representations as required by `avm_value_of_list` (Section 3.1.4.1 [Primitive types], page 72).

If the input parameters are inconsistent or if there is insufficient memory to allocate the result, the integer referenced by *fault* is assigned a non-zero value, and the list referenced by *message* is assigned a copy of the list representation of `<'bad matrix specification'>` or `<'memory overflow'>`, respectively. A non-empty message must be reclaimed by the caller using `avm_dispose` (Section 3.1.1 [Simple Operations], page 66).

If there are no errors, the result is a pointer to a packed array representation of the *operand* as explained above. The memory for this result is allocated by this function and should be freed by the caller when no longer required. The number of bytes allocated will be *item_size* * (*n* * (*n* + 1))/2.

`list` **avm_list_of_packed_matrix** (`int` *upper_trianguler*`,void *`*operand*`, int` *n*`, size_t` *item_size*`, int *`*fault*) — Function

This function performs an inverse operation to that of `avm_packed_matrix_of_list` given the address of a packed matrix stored according to one of the memory maps discussed above. The *operand* parameter holds the address, the parameter *n* gives the number of rows, and the *upper_triangular* parameter specifies which of the two possible memory maps to assume.

If there is sufficient memory, the result returned is a list in one of the triangular forms described above, being upper triangular if the *upper_triangular* parameter is non-zero, with values of length *item_size* taken from the array.

In the event of a memory overflow, the integer referenced by *fault* is assigned a non-zero value and the result is a copy of the message `<'memory overflow'>` represented as a list. A segmentation fault is possible if this function is passed an invalid pointer or dimension.

3.1.4.4 Related utility functions

A small selection of additional functions that are likely to be of use to developers concerned with matrix operations has been incorporated into the API to save the trouble of reinventing them, although doing so would be straightforward. They are described in this section without further motivation.

`void` ***avm_matrix_transposition** (`void *`*matrix*`, int` *rows*`, int` *cols*`, size_t` *item_size*) — Function

This function takes the address of an arbitrary rectangular *matrix* represented as a contiguous array (not a list) and transposes it in place. That is, this function transforms an *m* by *n* matrix to an *n* by *m* matrix by exchanging the *i*,*j*th element with the *j*,*i*th element for all values of *i* and *j*.

The numbers of rows and columns in the *matrix* are given by the parameters *rows* and *cols*, respectively, and the size of the entries in bytes is given by *item_size*.

The *matrix* is assumed to be in row major order, but this function is applicable to matrices in column major order if the caller passes the number of columns in *rows* and the number of rows in *cols*.

Alternatively, this function can be seen as a conversion between the row major and the column major representation of a matrix. An *m* by *n* matrix in row major order will be transformed to the same *m* by *n* matrix in column order, or from column order to row order.

A notable feature of this function is that it allocates no memory so there is no possibility of a memory overflow even for very large matrices, unlike a naive implementation which would involve making a temporary copy of the matrix. There is a possibility of a segmentation fault if invalid pointers or dimensions are given.

`void` ***avm_matrix_reflection** (`int` *upper_triangular*, `void *`*matrix*, `int` *n*, `size_t` *item_size*) — Function

This function takes a symmetric square *matrix* of dimension *n* containing entries of *item_size* bytes each and fills in the redundant entries.

If *upper_triangular* is non-zero, the upper triangle of the *matrix* is copied to the lower triangle. If *upper_triangular* is zero, the lower triangular entries are copied to the upper triangle.

These conventions assume row major order. If the *matrix* is in column major order, then the caller can either transpose it in place before and after this function by `avm_matrix_transposition`, or can complement the value of *upper_triangular*.

Note that this function may be unnecessary for `LAPACK` library functions that ignore the redundant entries in a symmetric matrix, because they can be left uninitialized, but it is included for the sake of completeness.

`list` ***avm_row_number_array** (`counter` *m*, `int *`*fault*) — Function

A fast, memory efficient finite map from natural numbers to their list representations can be obtained by using this function as an alternative to `avm_natural` or `avm_recoverable_natural` when repeated evaluations of numbers within a known range are required (Section 3.1.1 [Simple Operations], page 66 and Section 3.1.2 [Recoverable Operations], page 69).

Given a positive integer *m*, this function allocates and returns an array of *m* lists whose *i*th entry is the list representation of the number *i* as explained in Section 2.4 [Representation of Numeric and Textual Data], page 23.

An amount of memory proportional to *m* is used for the array and its contents. If there is insufficient memory, a `NULL` value is returned and the integer referenced by *fault* is set to a non-zero value.

`void` **avm_dispose_rows** (`counter` *m*, `list *`*row_number*) — Function

This function reclaims an array *row_number* of size *m* returned by `avm_row_number_array`, and its contents if any. A `NULL` pointer is allowed as the *row_number* parameter and will have no effect, but an uninitialized pointer will cause a segmentation fault.

`void` **avm_initialize_matcon** (); — Function

This function initializes some static variables used by the functions declared in ‘`matcon.h`’ and should be called before any of them is called or they might not perform according to specifications.

`void` **avm_count_matcon** `();` Function

This function frees the static variables allocated by `avm_initialize_matcon` and is used to verify the absence of memory leaks. It should be called after the last call to any functions in '`matcon.h`' but before `avm_count_lists` if the latter is being used (Section 3.1.1 [Simple Operations], page 66).

3.1.5 Comparison

The file '`compare.h`' contains a few function declarations pertaining to the computation of the comparison predicate described in Section 2.7.11.1 [Compare], page 46. Some of the work is done by static functions in '`compare.c`' that are not recommended entry points to the library.

`void` **avm_initialize_compare** `()` Function

This function should be called once before the first call to `avm_comparison`, as it initializes some necessary internal data structures.

`void` **avm_count_compare** `()` Function

This function can be used to check for memory leaks, by detecting unreclaimed storage at the end of a run. The data structures relevant to comparison that could be reported as unreclaimed are known as "decision" nodes, but these should always be handled properly by the library without intervention. If `avm_count_lists` is also being used, the call to this function must precede it.

`list` **avm_comparison** (`list` *operand*, `int` **fault*) Function

This function takes a list operand representing a pair of trees and returns a list representing the logical value of their equality. If the operand is `NULL`, a message of invalid comparison is returned and the **fault* is set to a non-zero value. If the `head` of the operand is unequal to the `tail`, a `NULL` value is returned. If they are equal, a list is returned whose `head` and `tail` are both `NULL`. The equality in question is structural rather than pointer equality.

The list operand to this function may be modified by this function, but not in a way that should make any difference to a client program. If two lists are found to be equal, or if even two sublists are found to be equal in the course of the comparison, one of them is deallocated and made to point to the other. This action saves memory and may make subsequent comparisons faster. However, it could disrupt client programs that happen to be holding stale list pointers.

As of `avram` version 0.6.0, a logical field called `discontiguous` has been added to the `node` record type declared in `lists.h`, which is checked by the comparison function. If a list node has its `discontiguous` field set to a non-zero value, and if it also has a non-null `value` field, then it won't be deallocated in the course of comparison even if it is found to be equal to something else. This feature can be used by client modules to create lists in which value fields refer to data structures that are meant to exist independently of them. See '`mpfr.c`' for an example.

This function is likely to have better performance and memory usage than a naive implementation of comparison, for the above reasons and also because of optimizations pertaining to comparison of lists representing characters. Moreover, it is not subject to stack overflow exceptions because it is not written in a recursive style.

`list` **avm_binary_comparison** (`list` *left_operand*, `list` *right_operand*, `int` **fault*); Function

This function is the same as `avm_comparison` except that it allows the left and right operands to be passed as separate lists rather than taking them from the `head` and the `tail` of a single list.

3.1.6 Deconstruction Functions

A fast native implementation of the deconstruction operation is provided by the functions declared in '`decons.h`'.

`void` **avm_initialize_decons** () Function

This should be called prior to the first call to `avm_deconstruction`, so as to initialize some necessary internal data structures. Results will be undefined if it is not.

`void` **avm_count_decons** () Function

For ecologically sound memory management, this function should be called at the end of a run to verify that there have been no leaks due to the deconstruction functions, which there won't be unless the code in '`decons.c`' has been ineptly modified. An error message to the effect of unreclaimed "points" could be the result otherwise.

`list` **avm_deconstruction** (`list` *pointer*, `list` *operand*, `int` **fault*) Function

Deconstructions are performed by this function, as described in Section 2.7.8.1 [Field], page 42. In the `silly` program notation (Section 2.7.4 [A Simple Lisp Like Language], page 35), this function computes the value of ([[`field`]] *pointer*) *operand*.

For example, using the fixed list `avm_join(NULL,NULL)` as the *pointer* parameter will cause a copy of the operand itself to be returned as the result. A *pointer* equal to `avm_join(NULL,avm_join(NULL,NULL))` will cause a copy of `operand->tail` to be returned, and so on. A `NULL` *pointer* causes an internal error.

If the deconstruction is invalid, as in the case of the tail of an empty list, the invalid deconstruction error message is returned as the result, and the **fault* parameter is set to a non-zero value. The **fault* parameter is also set to a non-zero value in the event of a memory overflow, and the memory overflow message is returned.

3.1.7 Indirection

In some cases it is necessary to build a tree from the top down rather than from the bottom up, when it is not known in advance what's on the bottom. Although the `list` type is a pointer itself, these situations call for a type of pointers to lists, which are declared as the `branch` type in '`branches.h`'. For example, if `b` is declared as a `branch` and `l` is declared as a `list`, it would be possible to write `b = &l`.

Facilities are also provided for maintaining queues of branches, which are declared as the `branch_queue` type. This type is a pointer to a structure with two fields, `above` and `following`, where `above` is a `branch` and `following` is a `branch_queue`.

These functions are used internally elsewhere in the library and might not be necessary for most client programs to use directly.

void **avm_initialize_branches** () Function

This must be done once before any of the other branch related functions is used, and creates some internal data structures. Results of the other functions are undefined if this one isn't called first.

void **avm_count_branches** () Function

This function can be used at the end of a run to detect unreclaimed storage used for branches or branch queues. If any storage remains unreclaimed, a message about unreclaimed branches is written to standard error.

void **avm_anticipate** (`branch_queue` **front*, `branch_queue` **back*, `branch` *operand*) Function

This function provides a simple queueing facility for branches. Similarly to the case with `avm_enqueue`, `front` and `back` should be initialized to `NULL` before the first call. Each call to this function will enqueue one item to the back, assuming enough memory is available, as the following example shows.

```
front = NULL;
back = NULL;
l = avm_join(NULL,NULL);
anticipate(&front,&back,&(l->head));
anticipate(&front,&back,&(l->tail));
```

After the above code is executed, these postconditions will hold.

```
front->above == &(l->head)
front->following->above == &(l->tail)
front->following == back
back->following == NULL
```

The name "anticipate" is used because ordinarily the queue contains positions in a tree to be filled in later. As usual, only unshared trees should be modified in place.

void **avm_recoverable_anticipate** (`branch_queue` **front*, `branch_queue` **back*, `branch` *operand*, `int` **fault*) Function

This function is similar to `avm_anticipate`, except that it will not exit with an error message in the event of an overflow error, but will simply set **fault* to a non-zero value and return to the caller. If an overflow occurs, nothing about the queue is changed.

void **avm_enqueue_branch** (`branch_queue` **front*, `branch_queue` **back*, `int` *received_bit*) Function

A slightly higher level interface to the `avm_anticipate` function is provided by this function, which is useful for building a tree from a string of input bits in a format similar to the one described in Section 2.2 [Concrete Syntax], page 20.

This function should be called the first time with `front` and `back` having been initialized to represent a queue containing a single branch pointing to a list known to the caller. The list itself need not be allocated or initialized. An easy way of doing so would be the following.

```
front = NULL;
back = NULL;
avm_anticipate(&front,&back,&my_list);
```

On each call to `avm_enqueue_branch`, the *received_bit* parameter is examined. If it is zero, nothing will be added to the queue, the list referenced by the front branch will be assigned `NULL`, and the front branch will be removed from the queue. If *received_bit* is a non-zero value, the list referenced by the front branch will be assigned to point to a newly created unshared list node, and two more branches will be appended to the queue. The first branch to be appended will point to the head of the newly created list node, and the second branch to be appended will point to the tail.

If the sequence of bits conforms to the required concrete syntax, this function can be called for each of them in turn, and at the end of the sequence, the queue will be empty and the list referenced by the initial branch (i.e., `my_list`) will be the one specified by the bit string. If the sequence of bits does not conform to the required concrete syntax, the error can be detected insofar as the emptying of the queue will not coincide exactly with the last bit.

The caller should check for the queue becoming prematurely empty due to syntax errors, because no message is reported by `avm_enqueue_branch` in that event, and subsequent attempts to enqueue anything are ignored. However, in the event of a memory overflow, an error message is reported and the process is terminated.

`void` **avm_recoverable_enqueue_branch** (`branch_queue` **front*, `branch_queue` **back*, `int` *received_bit*, `int` **fault*) Function

This function is similar to `avm_enqueue_branch` but will leave error handling to the caller in the event of insufficient memory to enqueue another branch. Instead of printing an error message and exiting, it will dispose of the queue, set the *fault* flag to a non-zero value, and return. Although the queue will be reclaimed, the lists referenced by the branches in it will persist. The list nodes themselves can be reclaimed by disposing of the list whose address was stored originally in the front branch.

`void` **avm_dispose_branch_queue** (`branch_queue` *front*) Function

This function deallocates a branch queue by chasing the `following` fields in each one. It does nothing to the lists referenced by the branches in the queue.

Rather than using `free` directly, client programs should use this function for deallocating branch queues, because it allows better performance by interacting with a local internal cache of free memory, and because it performs necessary bookkeeping for `avm_count_branches`.

`void` **avm_dispose_branch** (`branch_queue` *old*) Function

This disposes of a single branch queue node rather than a whole queue. Otherwise, the same comments as those above apply.

3.1.8 The Universal Function

A function computing the result of the invisible operator used to specify the virtual code semantics in Section 2.7 [Virtual Code Semantics], page 33, is easily available by way of a declaration in '`apply.h`'.

`void` **avm_initialize_apply** () — Function

This function should be called by the client program at least once prior to the first call to `avm_apply` or `avm_recoverable_apply`. It causes certain internal data structures and error message texts to be initialized.

`void` **avm_count_apply** () — Function

This function should be used at the end of a run for the purpose of detecting and reporting any unreclaimed storage associated with functions in this section. If the function `avm_count_lists()` is also being used, it should be called after this one.

`list` **avm_apply** (`list` *operator*, `list` *operand*) — Function

This is the function that evaluates the operator used to describe the virtual code semantics. For example, the value of $f\ x$ can be obtained as the result returned by `avm_apply(`*f*`,`*x*`)`.

Both parameters to this function are deallocated unconditionally and should not be referenced again by the caller. If the parameters are needed subsequently, then only copies of them should be passed to `avm_apply` using `avm_copied`.

This function is not guaranteed to terminate, and may cause a memory overflow error. In the event of an exceptional condition, the error message is written to standard error and the program is halted. There is no externally visible distinction between different levels of error conditions.

`list` **avm_recoverable_apply** (`list` *operator*, `list` *operand*, `int` `*`*fault*) — Function

This function is similar to `avm_apply` but leaves the responsibility of error handling with the caller. If any overflow or exceptional condition occurs, the result returned is a list representing the error message, and the *fault* flag is set to a non-zero value. This behavior contrasts with that of `avm_apply`, which will display the message to standard error and kill the process.

3.2 Characters and Strings

If a C program is to interact with a virtual code application by exchanging text, it uses the representation for characters described in Appendix A [Character Table], page 123. This convention would be inconvenient without a suitable API, so the functions in this section address the need. These functions are declared in the header file '`chrcodes.h`'.

Some of these functions have two forms, with one of them having the word `standard` as part of its name. The reason is to cope with multiple character encodings. Versions of `avram` prior to 0.1.0 used a different character encoding than the one documented in Appendix A [Character Table], page 123. The functions described in Section 3.5 [Version

Management], page 100 can be used to select backward compatible operation with the older character encoding. The normal forms of the functions in this section will use the older character set if a backward compatibility mode is indicated, whereas the standard forms will use the character encoding documented in Appendix A [Character Table], page 123 regardless.

Standard encodings should always be assumed for library and function names associated with the `library` combinator (Section 3.9.1 [Calling existing library functions], page 111), and for values of lists defined by `avm_list_of_value` (Section 3.1.4.1 [Primitive types], page 72), but version dependent encodings should be used for all other purposes such as error messages. Alternatively, the normal version dependent forms of the functions below can be used safely in any case if backward compatibility is not an issue. This distinction is viewed as a transitional feature of the API that will be discontinued eventually when support for the old character set is withdrawn and the `standard` forms are be removed.

`list` **avm_character_representation** (`int` *character*) Function

`list` **avm_standard_character_representation** (`int` *character*) Function

This function takes an integer character code and returns a copy of the list representing it, as per the table in Appendix A [Character Table], page 123. Because the copy is shared, no memory is allocated by this function so there is no possibility of overflow. Nevertheless, it is the responsibility of the caller dispose of the list when it is no longer needed by `avm_dispose`, just as if the copy were not shared (Section 3.1.1 [Simple Operations], page 66). For performance reasons, this function is implemented as a macro. If the argument is outside the range of zero to 255, it is masked into that range.

`int` **avm_character_code** (`list` *operand*) Function

`int` **avm_standard_character_code** (`list` *operand*) Function

This function takes a list as an argument and returns the corresponding character code, as per Appendix A [Character Table], page 123. If the argument does not represent any character, a value of `-1` is returned.

`list` **avm_strung** (`char` **string*) Function

`list` **avm_standard_strung** (`char` **string*) Function

This function takes a pointer to a null terminated character string and returns the list obtained by translating each character into its list representation and enqueuing them together. Memory needs to be allocated for the result, and if there isn't enough available, an error message is written to standard error and the process is terminated. This function is useful to initialize lists from hard coded strings at the beginning of a run, as in this example.

```
hello_string = avm_strung("hello");
```

This form initializes a single string, but to initialize a one line message suitable for writing to a file, it would have to be a list of strings, as in this example.

```
hello_message = avm_join(avm_strung("hello"),NULL);
```

The latter form is used internally by the library for initializing most of the various error messages that can be returned by other functions.

`list` **avm_recoverable_strung** (`char` **string*, `int` **fault*); Function

`list` **avm_recoverable_standard_strung** (`char` **string*, `int` **fault*); Function

This function is like `avm_strung` except that if it runs out of memory it sets the integer referenced by *fault* to a non-zero value and returns instead of terminating the process.

`char` ***avm_unstrung** (`list` *string*, `list` **message*, `int` **fault*) Function

`char` ***avm_standard_unstrung** (`list` *string*, `list` **message*, `int` **fault*) Function

This function performs an inverse operation to `avm_recoverable_strung`, taking a list representing a character string to the character string in ASCII null terminated form as per the standard C representation. Memory is allocated for the result by this function which should be freed by the caller.

In the event of an exception, the integer referenced by `fault` is assigned a non-zero value and an error message represented as a list is assigned to the list referenced by `message`. The error message should be reclaimed by the caller with `avm_dispose` (Section 3.1.1 [Simple Operations], page 66 if it is non-empty. Possible error messages are `<'memory overflow'>`, `<'counter overflow'>`, and `<'invalid text format'>`.

`list` **avm_scanned_list** (`char` **string*) Function

An application that makes use of virtual code snippets or data that are known at compile time can use this function to initialize them. The argument is a string in the format described in Section 2.2 [Concrete Syntax], page 20, and the result is the list representing it. For example, the program discussed in Section 1.8 [Example Script], page 15 could be hard coded into a C program by pasting the data from its virtual code file into an expression of this form.

```
cat_program = avm_scanned_list("sKYQNTP\\");
```

Note that the backslash character in the original data has to be preceded by an extra backslash in the C source, because backslashes usually mean something in C character constants.

The `avm_scanned_list` function needs to allocate memory. If there isn't enough memory available, it writes a message to standard error and causes the process to exit.

`list` **avm_multiscanned** (`char` ***strings*) Function

Sometimes it may be useful to initialize very large lists from strings, but some C compilers impose limitations on the maximum length of a string constant, and the ISO standard for C requires only 512 bytes. This function serves a similar purpose to `avm_scanned_list`, but allows the argument to be a pointer to a null terminated array of strings instead of one long string, thereby circumventing this limitation in the compiler.

```
char *code[] = {"sKYQ","NTP\\",NULL};
...
cat_program = avm_multiscanned(code);
```

If there is insufficient memory to allocate the list this function needs to create, it causes an error message to be written to standard error, and then kills the process.

`char*` **avm_prompt** (`list` *prompt_strings*) Function

This function takes a list representing a list of character strings, and returns its translation to a character string with the sequence 13 10 used as a separator. For example, given a tree of this form

```
some_message = avm_join(
   avm_strung("hay"),
   avm_join(
      avm_strung("you"),
      NULL));
```

the result returned by `prompt_strings(some_message)` would be a pointer to a null terminated character string equivalent to the C constant `"hay\13\10you"`.

Error messages are printed and the process terminated in the event of either a memory overflow or an invalid character representation.

This function is used by `avram` in the evaluation of interactive virtual code applications, whose output has to be compared to the output from a shell command in this format. The separator is chosen to be compatible with the `expect` library.

`char*` **avm_recoverable_prompt** (`list` *prompt_strings*, `list` `*`*message*, `int` `*`*fault*) Function

This function performs the same operation as `avm_prompt` but allows the caller to handle exceptional conditions. If an exception such as a memory overflow occurs, the integer referenced by `fault` is assigned a non-zero value and a representation of the error message as a list of strings is assigned to the list referenced by `message`.

This function is used to by `avram` to evaluate the `interact` combinator (Section 2.7.16.3 [Interaction combinator], page 61), when terminating in the event of an error would be inappropriate.

`void` **avm_initialize_chrcodes** () Function

This function has to be called before any of the other character conversion functions in this section, or else their results are undefined. It performs the initialization of various internal data structures.

`void` **avm_count_chrcodes** () Function

This function can be called at the end of a run, after the last call to any of the other functions in this section, but before `avm_count_lists` if that function is also being used. The purpose of this function is to detect and report memory leaks. If any memory associated with any of these functions has not been reclaimed by the client program, a message giving the number of unreclaimed lists will be written to standard error.

3.3 File Manipulation

The functions described in this section provide an interface between virtual code applications and the host file system by converting between files or file names and their representations as lists. These conversions are necessary when passing a file to a virtual code application, or when writing a file received in the result of one.

3.3.1 File Names

A standard representation is used by virtual code applications for the path names of files, following the description in Section 2.6.1 [Input Data Structure], page 27. The functions and constants declared in `fnames.h` provide an API for operating on file names in this form.

`list` **avm_path_representation** (`char` **path*) Function

If a C program is to invoke a virtual code application and pass a path name to it as a parameter, this function can be used to generate the appropriate representation from a given character string.

```
conf_path = avm_path_representation("/etc/resolve.conf");
```

In this example, `conf_path` is a `list`. For potentially better portability, a C program can use the character constant `avm_path_separator_character` in place of the slashes in hard coded path names.

Other useful constants are `avm_current_directory_prefix` as a portable replacement for `"./"`, as well as `avm_parent_directory_prefix` instead of `"../"`. There is also `avm_root_directory_prefix` for `"/"`. These three constants are null terminated strings, unlike `avm_path_separator_character`, which is a character.

If a `NULL` pointer is passed as the *path*, a `NULL` list is returned, which is the path representation for standard input or standard output. If the address of an empty string is passed to this function as the *path*, the list of the empty string will be returned, which is the path representation for the root directory. Trailing path separators are ignored, so `"/"` is the same as the empty string.

Some memory needs to be allocated for the result of this function. If the memory is not available, an error message is written to standard error and the process is terminated.

`list` **avm_date_representation** (`char` **path*) Function

This function is essentially a wrapper around the standard `ctime_r` function that not only gets the time stamp for a file at a given path, but transforms it to a list representation according to Appendix A [Character Table], page 123. It needs to allocate memory for the result and will cause the program to exit with an error message if there is not enough memory available.

The time stamp will usually be in a format like `Sun Mar 4 10:56:40 GMT 2001`. If for some reason the time stamp can not be obtained, the result will be a representation of the string `unknown date`.

`char*` **avm_path_name** (`list` *path*) Function

This function is the inverse of `avm_path_representation`, in that it takes a list representing a path to the path name expressed as a character string. This function can be used in C programs that invoke virtual code applications returning paths as part of their results, so that the C program can get the path into a character string in order to open the file.

If the *path* parameter is `NULL`, a `NULL` pointer is returned as the result. The calling program should check for a `NULL` result and interpret it as the path to standard input or standard output.

The memory needed for the character string whose address is returned is allocated by this function if possible. The given *path* is not required to be consistent with the host file system, but it is required to consist of representations of non-null printable characters or spaces as lists per Appendix A [Character Table], page 123. In the event of any error or overflow, control does not return to the caller, but an error message is printed and the program is aborted. The possible error messages from this function are the following.

- *program-name*`: counter overflow (code` *nn*`)`
- *program-name*`: memory overflow (code` *nn*`)`
- *program-name*`: null character in file name`
- *program-name*`: bad character in file name`
- *program-name*`: invalid file name (code` *nn*`)`

`void` **avm_initialize_fnames** () Function

A few housekeeping operations relevant to internal data structures are performed by this function, making it necessary to be called by the client program prior to using any of the other ones.

`void` **avm_count_fnames** () Function

This function can be used after the the last call to any of the other functions in this section during a run, and it will detect memory leaks that may be attributable to code in these functions or misuse thereof. If any unreclaimed storage remains when this function is called, a warning message will be written to standard error. If the function `avm_count_lists` is also being used by the client, it should be called after this one.

3.3.2 Raw Files

Some low level operations involving lists and data files are provided by these functions, which are declared in the header file '`rawio.h`'.

`list` **avm_received_list** (`FILE *`*object*`, char *`*filename*) Function

This function is a convenient way of transferring data directly from a raw format file into a list in memory. It might typically be used to load the virtual code for an application that has been written to a file by a compiler.

object is the address of a file which should already be open for reading before this function is called, and will be read from its current position.

filename should be set by the caller to the address of a null terminated string containing the name of the file, but is not used unless it needs to be printed as part of an error message. If it is a null pointer, standard input is assumed.

The result returned is a list containing data read from the file.

The file format is described in Section 2.3 [File Format], page 22. The preamble section of the file, if any, is ignored. If the file ends prematurely or otherwise conflicts with the format, the program is aborted with a message of

program-name`: invalid raw file format in` *filename*

written to standard error. The program will also be aborted by this function in the event of a memory overflow.

The file is left open when this function returns, and could therefore be used to store other data after the end of the list. The end of a list is detected automatically by this function, and it reads no further, leaving the file position on the next character, if any.

`void` **avm_send_list** (`FILE *`*repository*`, list` *operand*`, char *`*filename*) Function

This function can be used to transfer data from a list in memory to a file, essentially by implementing the printing algorithm described in Section 2.2.1 [Bit String Encoding], page 21.

repository is the address of a file already open for writing, to which the data are written starting from the current position.

operand is the list containing the data to be written

filename is the address of a null terminated string containing the name of the file that will be reported in an error message if necessary.

No preamble section is written by this function, but one could be written to the file by the caller prior to calling it. Error messages are possible either because of i/o errors or because of insufficient memory. I/o errors are not fatal and will result only in a warning message being printed to standard error, but a memory overflow will cause the process to abort. An i/o error message reported by this function would be of the form

program-name`: can't write to` *filename*

followed by the diagnostic obtained from the standard `strerror` function if it exists on the host platform. The file is left open when this function returns.

`void` **avm_initialize_rawio** () Function

This function initializes some necessary data structures for the functions in this section, and should be called prior to them at the beginning of a run.

`void` **avm_count_rawio** () Function

This function does nothing in the present version of the library, but should be called after the last call to all of the other functions in this section in order to maintain compatibility with future versions of the library. Future versions may decide to use this function to do some cleaning up of local data structures.

3.3.3 Formatted Input

Some functions relating to the input of text files or data files with preambles are declared in the header file '`formin.h`'. The usage of these functions is as follows.

`list` **avm_preamble_and_contents** (`FILE *`*source*, `char *`*filename*) Function

This function loads a file of either text or data format into memory.

source should be initialized by the caller as the address of a file already open for reading that will be read from its current position.

filename should be set by the caller to the address of a null terminated character string giving the name of the file that will be used if an i/o error message needs to be written about it. If it is a `NULL` pointer, standard input is assumed.

The result returned by the function will be a list whose `head` represents the preamble of the file and whose `tail` represents the contents. As a side effect, the input file will be closed, unless the *filename* parameter is `NULL`.

If the file conforms to the format described in Section 2.3 [File Format], page 22, the preamble is a list of character strings. In the result returned by the function, the `head` field will be a list with one item for each line in the file, and each item will be a list of character representations as in Appendix A [Character Table], page 123, but with the leading hashes stripped. The `tail` will be the list specified by remainder of the file according to Section 2.2 [Concrete Syntax], page 20. If the file has an empty preamble but is nevertheless a data file, the `head` will be a list whose `head` and `tail` are both `NULL`.

If the file does not conform to the format in Section 2.3 [File Format], page 22, then the `head` of the result will be `NULL`, and the `tail` will be a list of lists of character representations, with one for each line.

Whether or not the file conforms to the format is determined on the fly, so this function is useful for situations in which the format is not known in advance. The conventions regarding the preamble and contents maintained by this function are the same as those used by virtual code applications as described in Section 2.5.1.2 [Standard Output Representation], page 25 and Section 2.6.1 [Input Data Structure], page 27.

The characters used for line breaks are not explicitly represented in the result. Depending on the host system, line breaks in text files may be represented either by the character code 10, or by the sequence 13 10. However, in order for the library to deal with binary files in a portable way, a line break always corresponds to a 10 as far as this function is concerned regardless of the host, and a 13 is treated like any other character. Hence, if this function were used on binary files that happened to have some 10s in them, the exact contents of a file could be reconstructed easily by appending a 10 to all but the last line and flattening the list.

A considerable amount of memory may need to be allocated by this function in order to store the file as a list. If not enough memory is available, the function prints an error message to standard error and aborts rather than returning to the caller.

However, i/o errors are not fatal, and will cause the function to print a warning but attempt to continue.

`list` **avm_load** (`FILE` `*`*source*`,` `char` `*`*filename*`,` `int` *raw*) Function

Similarly to `avm_preamble_and_contents`, this function also loads a file into memory, but the format is specified in advance.

source should be set by the caller to the address of an already open file for reading, which will be read from its current position.

filename should be initialized by the caller as a pointer to a null terminated string containing the name of the file that will be reported to the user in the event of an error reading from it. If it is a `NULL` pointer, standard input is assumed.

raw is set to a non-zero value by the caller to indicate that the file is expected to conform to the format in Section 2.3 [File Format], page 22. If the file is an ordinary text file, then it should be set to zero.

In the case of a data file, which is when *raw* is non-zero, the result returned by this function will be a list representing the data section of the file and ignoring the preamble. In the case of a text file, the result will be a list of lists of character representations as per Appendix A [Character Table], page 123, with one such list for each line in the file. Similar comments about line breaks to those mentioned under `avm_preamble_and_contents` are applicable.

As a side effect of this function, the *source* file will be closed, unless the *filename* is a `NULL` pointer.

This function is useful when the type of file is known in advance. If a data file is indicated by the *raw* parameter but the format is incorrect, an error message is reported and the process terminates. The error message will be of the form

program-name`: invalid raw file format in` *filename*

Alternatively, if a text file is indicated by the *raw* parameter, then no attempt is made to test whether it could be interpreted as data, even if it could be. This behavior differs from that of `avm_preamble_and_contents`, where a bad data file format causes the file to be treated as text, and a valid data file format, even in a "text" file, causes it to be treated as data.

Memory requirements for this function are significant and will cause the process to abort with an error message in the event of insufficient free memory. Messages pertaining to i/o errors are also possible and are not fatal.

`void` **avm_initialize_formin** () Function

This function should be called before either of the other functions in this section is called, as it initializes some necessary static data structures. Results of the other functions are undefined if this one is not called first.

`void` **avm_count_formin** () Function

This function should be called after the last call to any of the other functions in this section, as it is necessary for cleaning up and reclaiming some internal data. If any storage remains unreclaimed due to memory leaks in these functions or to misuse of them, a warning message is written to standard error. If the function `avm_count_lists` is also being used by the client program, it should be called after this one.

3.3.4 Formatted Output

The following functions pertaining to the output of text files or data files with preambles are declared in the header file '`formout.h`'.

`void` **avm_output** (`FILE *`*repository*`, char *`*filename*`, list` *preamble*`, list` *contents*`, int` *trace_mode*) Function

This function writes a either a text file or a data file in the format described in Section 2.3 [File Format], page 22. The parameters have these interpretations.

repository is the address of a file opened for writing by the caller, that will be written from its current position.

filename is the address of a null terminated character string set by the caller to be the name of the file that will be reported to the user in the event of an i/o error.

preamble is `NULL` in the case of a text file, but a list of character string representations as per Appendix A [Character Table], page 123, in the case of a data file. If a data file has is to be written with an empty preamble, then this list should have a `NULL` `head` and a `NULL` `tail`.

contents is either a list of character string representations in the case of a text file, or is an unconstrained list in the case of a data file.

trace_mode
may be set to a non-zero value by the caller to request that everything written to a text file should be echoed to standard output. It is ignored in the case of a data file.

The effect of calling this function is to write the preamble and contents to the file in the format indicated by the preamble. The file is left open when this function returns.

Line breaks are always written as character code 10, not as 13 10, regardless of the convention on the host system, so that files written by this function can be reliably read by other functions in the library.

Leading hashes are automatically added to the beginning of the lines in the preamble, except where they are unnecessary due to a continuation character on the previous line. This action enforces consistency with the file format, ensuring that anything written as a data file can be read back as one. The hashes are stripped automatically when the file is read by `avm_preamble_and_contents`.

Another feature of this function is that it will mark any output file as executable if it is a data format file with a prelude whose first character in the first line is an exclamation point. This feature makes it easier for a compiler implemented in virtual code to generate executable shell scripts directly.

A fatal error is reported if any of the data required to be a character representation is not listed in the Appendix A [Character Table], page 123. A fatal error can also be caused by a memory overflow. Possible error messages are the following.

- *program-name*: `invalid output preamble format`
- *program-name*: `invalid text format`
- *program-name*: `can't write to` *filename*

In the last case, the error message will be followed by an explanation furnished by the standard `strerror` function if available.

`void` **avm_output_as_directed** (`list` *data*, `int` *ask_to_overwrite_mode*, `int` *verbose_mode*) Function

This function writes an ensemble of files at specified paths in either text or data format, optionally interacting with the user through standard input and output. The parameters have these interpretations.

data is a list in which each item specifies a file to be written.

ask_to_overwrite_mode
: may be set to a non-zero value by the calling program in order to have this function ask the user for permission to overwrite existing files.

verbose_mode
: may be set to a non-zero value by the calling program to have this function print to standard output a list of the names of the files it writes.

A high level interface between virtual code applications and the file system is provided by this function. The *data* parameter format is compatible with the the data structure returned by an application complying with the conventions in Section 2.6.3 [Output From Non-interactive Applications], page 30.

Each item in the *data* list should be a non-empty list whose `head` and `tail` are also non-empty. The fields in each item have the following relevance to the file it specifies.

- The `head` of the `head` is `NULL` if the file is to be opened for appending, and non-`NULL` if it is to be overwritten.
- The `tail` of the `head` represents a path as a list of character string representations, in a form suitable as an argument to `avm_path_name`.
- The `head` of the `tail` represents the preamble of the file, as either `NULL` for a text file or a non-empty list of character string representations for a data file.
- The `tail` of the `tail` represents the contents of the file, either as a list of character string representations for a text file or as a list in an unconstrained format for a data file.

For each item in the list, the function performs the following steps.

1. It decides whether to open a file for overwriting or appending based on the `head` of the `head`.
2. It uses the `tail` of the `head` to find out the file name from `avm_path_name`, in order to open it.
3. If the *ask_to_overwrite_mode* flag is set and the file is found to exist already, the function will print one of the following messages to standard output, depending on whether the file is to be overwritten or appended.

  - *program-name*: `overwrite` *filename*`? (y/n)`
  - *program-name*: `append to` *filename*`? (y/n)`

   It will then insist on either *y* or *n* as an answer before continuing.

4. If the *ask_to_overwrite* flag has not been set, or the file did not previously exist, or the answer of *y* was given, the preamble and contents of the file are then written with `avm_output`.
5. If permission to write or append was denied, one of the following messages is reported to standard output, and the data that were to be written are lost.
   - *program-name*: `not writing` *filename*
   - *program-name*: `not appending` *filename*
6. If permission was granted to write or append to the file or the *verbose_mode* flag is set, one of the messages
   - *program-name*: `writing` *filename*
   - *program-name*: `appending` *filename*

   is sent to standard output.

If any files are to be written to standard output, which would be indicated by a `NULL` path, they are not written until all other files in the list are written. This feature is in the interest of security, as it makes it more difficult for malicious or inept virtual code to alter the appearance of the console through standard output until after the interactive dialogue has taken place. Permission is not solicited for writing to standard output, and it will not be closed.

Any of the fatal errors or i/o errors possible with `avm_output` or `avm_path_name` are also possible with this function, as well as the following additional ones.

- *program-name*: `invalid file specification`
- *program-name*: `can't close` *filename*
- *program-name*: `can't write` *filename*

The last two are non-fatal i/o errors that will be accompanied by an explanation from the `strerror` function if the host supports it. The other message is the result of a badly formatted *data* parameter.

`void` **avm_put_bytes** (`list` *bytes*) Function

This function takes a list of character representations, converts them to characters, and sends them to standard output. There is no chance of a memory overflow, but the following other errors are possible.

- *program-name*: `invalid text format (code` *nn*`)`
- *program-name*: `can't write to standard output`

The latter is non-fatal, but the former causes the program to abort. It is caused when any member of the list *bytes* is not a character representation appearing in Appendix A [Character Table], page 123.

`void` **avm_initialize_formout** () Function

This function initializes some data structures used locally by the other functions in this section, and should be called at the beginning of a run before any of them is called.

`void` **avm_count_formout** () Function

This function doesn't do anything in the current version of the library, but should be called after the last call to any of the other functions in this section. Future versions of the library might use this function for cleaning up some internal data structures, and client programs that call it will maintain compatibility with them.

3.4 Invocation

The functions documented in this section can be used to incorporate the capabilities of a virtual machine emulator into other C programs with a minimal concern for the details of the required data structures and virtual code invocation conventions.

3.4.1 Command Line Parsing

A couple of functions declared in '`cmdline.h`' can be used to do all the necessary parsing of command lines and environment variables needed by virtual code applications.

`list` **avm_default_command_line** (`int` *argc*, `char *`*argv*[], `int` *index*, `char *`*extension*, `char *`*paths*, `int` *default_to_stdin_mode*, `int` *force_text_input_mode*, `int *`*file_ordinal*) Function

The purpose of this function is to build most of the data structure used by parameter mode applications, as described in Section 2.6.1 [Input Data Structure], page 27, by parsing the command line according to Section 1.5 [Command Line Syntax], page 8. The parameters have these interpretations.

argc is the number elements in the array referenced by *argv*

argv is the address of an array of pointers to null terminated character strings holding command line arguments

index is the position of the first element of *argv* to be considered. Those preceding it are ignored.

extension is the address of a string that will be appended to input file names given in *argv* in an effort to find the associated files

paths is the address of a null terminated character string containing a colon separated list of directory names that will be searched for input files

default_to_stdin_mode
is set to a non-zero value by the caller if the contents of standard input should be read in the absence of input files

force_text_input_mode
is set to a non-zero value by the caller to indicate that input files should be read as text, using `avm_load` (rather than `avm_preamble_and_contents`,

which would allow them to be either text or data). The *preamble* field of the returned file specifications will always be empty when this flag is set.

file_ordinal
: is set to a pointer to an integer by the caller if only one file is to be loaded during each call. The value of the integer indicates the which one it will be.

The result returned by this function is a list whose `head` is a list of file specifications and whose `tail` is a list of command line options intended for input to a virtual code application.

The list of file specifications returned in the `head` of the result follows the same conventions as the *data* parameter to the function `avm_output_as_directed`, except that the `head` of the `head` of each item is a list representing the time stamp of the file as given by `avm_date_representation`. If the file is standard input, then it holds the current system date and time.

If the *file_ordinal* parameter is `NULL`, then all files on the command line are loaded, but if it points to an integer n, then only the nth file is loaded, and n is incremented. If there is no nth file, a `NULL` value is returned as the entire result of the function. For a series of calls, the integer should be initialized to zero by the caller before the first call.

If standard input is indicated as one of the files on the command line (by a dash), then it is also loaded regardless of the *file_ordinal*, but a cached copy of it is used on subsequent calls after the first, so that the function does not actually attempt to reread it. If standard input is to be loaded, it must be finite for this function to work properly.

The search strategy for files is described in Section 1.10 [Environment], page 16, and makes use of the *extension* and *paths* parameters.

In the list of command line options returned in the `tail` of the result, each item is a list with a non-empty `head` and `tail`, and is interpreted as follows.

- The `head` of the `head` is a list representing a natural number, as given by `avm_natural`, indicating the position of the option on the command line relative to the initial value of the *index* parameter.
- The `tail` of the `head` is a list which is `NULL` in the case of a "short form" option, written with a single dash on the command line, but is a list whose `head` and `tail` are `NULL` in the case of a "long form" option, written with two dashes.
- The `head` of the `tail` is a list representing a character string for the keyword of an option, for example *`foo`* in the case of an option written *`--foo=bar,baz`*.
- The `tail` of the `tail` is a list of lists representing character strings, with one item for each parameter associated with the option, for example, *`bar`* and *`baz`*.

If multiple calls to the function are made with differing values of `*`*file_ordinal* but other parameters unchanged, the same list of options will be returned each time, except insofar as the position numbers in the `head` of the `head` of each item are adjusted as explained in Section 2.6.2 [Input for Mapped Applications], page 29.

Any of the i/o errors or fatal errors associated with other file input operations are possible with this function as well. This non-fatal warning message is also possible.

program-name: `warning: search paths not supported`

This error occurs if the library has been built on a platform that doesn't have the '`argz.h`' header file and the *paths* parameter is non-`NULL`.

`list` **avm_environment** (`char` **env*[]) Function

This function takes the address of a null terminated array of pointers to null terminated character strings of the form `"variable=value"`. The result returned is a list of lists, with one item for each element of the array. The `head` of each item is a representation of the left side of the corresponding string, and the `tail` is a representation of the right.

This function is therefore useful along with `avm_default_command_line` for building the remainder of the data structure described in Section 2.6 [Parameter Mode Interface], page 27. For example, a virtual machine emulator for non-interactive parameter mode applications with no bells and whistles could have the following form.

```
int
main(argc,argv,env)
...
{
  FILE *virtual_code_file;
...
  avm_initialize_lists();
  avm_initialize_apply();
  avm_initialize_rawio();
  avm_initialize_formout();
  avm_initialize_cmdline();
  virtual_code_file = fopen(argv[1],"rb");
  operator = avm_received_list(
    virtual_code_file,argv[1]);
  fclose(virtual_code_file);
  command = avm_default_command_line(argc,
    argv,2,NULL,NULL,0,0,NULL);
  environs = avm_environment(env);
  operand = avm_join(command,environs);
  result = avm_apply(operator,operand);
  avm_output_as_directed(result,0,0);
  avm_dispose(result);
...
}
```

The `avm_environment` function could cause the program to abort due to a memory overflow. For security reasons, it will also abort with an error message if any non-printing characters are detected in its argument. (See Section 1.6.7 [Other Diagnostics and Warnings], page 14.)

`void` **avm_initialize_cmdline** () Function

This function initializes some local variables and should be called before any of the other functions in this section is called, or else their results are unpredictable.

`void` **avm_count_cmdline** () Function

This function should be called after the last call to any of the other functions in this section, as it reclaims some locally allocated storage. If the `avm_count_lists` function is used, it should be called after this one.

3.4.2 Execution Modes

Some functions declared in '`exmodes.h`' are useful for executing interactive applications or filter mode transducers in a manner consistent with the specifications described in the previous chapter.

`void` **avm_interact** (`list` *avm_interactor*, `int` *step_mode*, `int` *ask_to_overwrite_mode*, `int` *quiet_mode*) Function

This function executes an interactive virtual code application. The parameters have these interpretations.

avm_interactor
: is the virtual code for a function that performs as specified in Section 2.6.4 [Output From Interactive Applications], page 30.

step_mode
: will cause all shell commands to be echoed if set to a non-zero value, and will cause the program to pause after each shell command until a key is pressed.

ask_to_overwrite_mode
: can be set to a non-zero value by the caller to cause the program to ask permission of the user to overwrite any existing files in cases where the virtual code returns a file list as described in Section 2.6.4.3 [Mixed Modes of Interaction], page 33.

quiet_mode
: can be set to a non-zero value to suppress console messages in the case of file output per Section 2.6.4.3 [Mixed Modes of Interaction], page 33.

The meaning of this function is accessible to any reader willing to slog through Section 2.6.4 [Output From Interactive Applications], page 30. The only subtle point is that *avm_interactor* parameter in this function does not correspond to the virtual code application that `avram` reads from a virtual code file, but to the result computed when the application read from the file is applied to the data structure representing the command line and environment.

Any of the memory overflows or i/o errors possible with other functions in the library are possible from this one as well, and will also cause it to print an error message and halt the program. A badly designed virtual code application could cause a deadlock, which will not be detected or reported

`void` **avm_trace_interaction** () Function

This function enables diagnostic output for the `avm_recoverable_interact` function.

`void` **avm_disable_interaction** () Function

This function causes `avm_interact` and `avm_recoverable_interact` to terminate with an error instead of executing, as required by the `--jail` command line option.

`list` **avm_recoverable_interact** (`list` *interactor*, `int` **fault*) Function

This function is similar to `avm_interact` but always closes the pipe and performs no file i/o, and will return an error message rather than exiting. Otherwise it returns a transcript of the intereaction as a list of lists of strings represented as lists of character encodings. It implements the *interact* combinator with the virtual code for the transducer function given as the parameter. A prior call to `avm_trace_interaction` will cause diagnostic information to be written to standard output when this function is executed.

`void` **avm_byte_transduce** (`list` *operator*) Function

This function executes a filter mode byte transducer application, which behaves as described in Section 2.5.3 [Byte Transducers], page 26. The argument is the virtual code for the application, which would be found in a virtual code file. There are limited opportunities for i/o errors, as only standard input and standard output are involved with this function, but fatal errors due to memory overflow are possible.

`void` **avm_line_map** (`list` *operator*) Function

This function executes line mapped filter mode applications, which are explained in Section 2.5.2 [Line Maps], page 26. The argument is the virtual code for the application. Similar comments to those above apply.

`void` **avm_initialize_exmodes** () Function

This function should be called before any of the other functions in this section in order to initialize some local variables. Results are undefined if this function isn't called first.

`void` **avm_count_exmodes** () Function

This function doesn't do anything in the present version of the library, but should be called after the last call to any of the other functions in this section in order to maintain compatibility with future versions, which may use it for cleaning up local variables.

3.5 Version Management

The `avram` library is designed to support any number of backward compatibility modes with itself, by way of some functions declared in '`vman.h`'. The assumption is that the library will go through a sequence of revisions during its life, each being identified by a unique number. In the event of a fork in the project, each branch will attempt to maintain compatibility at least with its own ancestors.

`void` **avm_set_version** (`char` **number*) Function

This function can be used to delay the demise of a client program that uses the library but is not updated very often. The argument is a null terminated string representing a version number, such as `"0.13.0"`.

A call to this function requests that all library functions revert to their behavior as of that version in any cases where the current behavior is incompatible with it. It will also cause virtual code applications evaluated with `avm_apply` to detect a version number equal to the given one rather than the current one. (See Section 2.7.7.1 [Version], page 41.)

The program will exit with an internal error message if any function in the library has already interrogated the version number before this function is called, or if it is passed a null pointer. This problem can be avoided by calling it prior to any of the `avm_initialize` functions with a valid address. The program will exit with the message

program-name`: multiple version specifications`

if this function is called more than once, even with the same number. If the number is not recognized as a present or past version, or is so old that it is no longer supported, the program will exit with this message.

`avram: can't emulate version` *number*

Client programs that are built to last should allow the version number to be specified as an option by the user, and let virtual code applications that they execute take care of their own backward compatibility problems. This strategy will at least guard against changes in the virtual machine specification and other changes that do not affect the library API.

`int` **avm_prior_to_version** (`char *`*number*) Function

This function takes the address of a null terminated string representing a version number as an argument, such as `"0.13.0"`, and returns a non-zero value if the version currently being emulated predates it.

If no call has been made to `avm_set_version` prior to the call to this function, the current version is assumed, and subsequent calls to `avm_set_version` will cause an internal error.

The intended use for this function would be by a maintainer of the library introducing an enhancement that will not be backward compatible, who doesn't wish to break existing client programs and virtual code applications. For example, if a version `1.0` is developed at some time in the distant future, and it incorporates a previously unexpected way of doing something, code similar to the following could be used to maintain backward compatibility.

```
if (avm_prior_to_version("1.0"))
  {
    /* do it the 0.x way */
  }
else
  {
    /* do it the 1.0-and-later way */
  }
```

This function will cause an internal error if the parameter does not match any known past or present version number, or if it is a null pointer.

`char*` **avm_version** () Function

This function returns the number of the version currently being emulated as the address of a null terminated string. The string whose address is returned should not be modified by the caller.

If no call has been made to `avm_set_version` prior to the call to this function, the current version is assumed, and subsequent calls to `avm_set_version` will cause an internal error.

3.6 Error Reporting

Most of the error reporting by other functions in the library is done by way of the functions declared in '`error.h`'. These function communicate directly with the user through standard error. Client programs should also use these functions where possible for the sake of a uniform interface.

`void` **avm_set_program_name** (`char *`*argv0*) Function

The argument to this function should be the address of a null terminated string holding the name of the program to be reported in error messages that begin with a program name. Typically this string will be the name of the program as it was invoked on the command line, possibly with path components stripped from it. An alternative would be to set it to the name of a virtual code application being evaluated. If this function is never called, the name `"avram"` is used by default. Space for a copy of the program name is allocated by this function, and a fatal memory overflow error is possible if there is insufficient space available.

`char*` **avm_program_name** () Function

This function returns a pointer to a null terminated character string holding the program name presently in use. It will be either the name most recently set by `avm_set_program_name`, or the default name `"avram"` if none has been set. The string whose address is returned should not be modified by the caller.

`void` **avm_warning** (`char *`*message*) Function

This function writes the null terminated string whose address is given to standard error, prefaced by the program name and followed by a line break.

`void` **avm_error** (`char *`*message*) Function

This function writes the null terminated string whose address is given to standard error, prefaced by the program name and followed by a line break, as `avm_warning`, but it then terminates the process with an exit code of 1.

`void` **avm_fatal_io_error** (`char *`*message*, `char *`*filename*, `int` *reason*) Function

This function is useful for reporting errors caused in the course of reading or writing files. The message is written to standard error prefaced by the program name, and incorporating the name of the relevant file. The *reason* should be the error code obtained from the standard `errno` variable, which will be translated to an informative message if possible by the standard `strerror` function and appended to the message. After the message is written, the process will terminate with an exit code of 1.

`void` **avm_non_fatal_io_error** (`char *`*message*, `char *`*filename*, `int` *reason*) Function

This function does the same as `avm_fatal_io_error` except that it doesn't exit the program, and allows control to return to the caller, which should take appropriate action.

`void` **avm_internal_error** (`int` *code*) Function

This function is used to report internal errors and halt the program. The error message is written to standard error prefaced by the program name and followed by a line break. The code should be a unique integer constant (i.e., not one that's used for any other internal error), that will be printed as part of the error message as an aid to the maintainer.

This function should be used by client programs only in the event of conditions that constitute some violation of a required invariant. It indicates to the user that something has gone wrong with the program, for which a bug report would be appropriate.

`void` **avm_reclamation_failure** (`char *`*entity*, `counter` *count*) Function

This function is used only by the `avm_count` functions to report unreclaimed storage. The *count* is the number of units of storage left unreclaimed, and the *entity* is the address of a null terminated string describing the type of unreclaimed entity, such as `"lists"` or `"branches"`. The message is written to standard error followed by a line break, but the program is not halted and control returns to the caller.

3.7 Profiling

The functions declared in '`profile.h`' can be used for constructing and writing tables of run time statistics such as those mentioned in Section 1.9 [Files], page 15, and Section 2.7.7.3 [Profile], page 41. These functions maintain a database of structures, each recording the statistics for a particular virtual code fragment.

Each structure in the database is identified by a unique key, which must be a list representing a character string. A pointer to such a structure is declared to be of type `score`. For the most part, the data structure should be regarded as opaque by a client program, except for a field `reductions` of type `counter`, which may be modified arbitrarily by the client.

The way these operations are used in the course of evaluating virtual code applications containing profile annotations is to add a structure to the database each time a new profiled code fragment is encountered, using the annotation as its key, and to increment the `reductions` of the structure each time any constituent of the code gets a quantum of work done on it. Other ways of using these operations are left to the developer's discretion.

`score` **avm_entries** (`list` *team*, `list *`*message*, `int *`*fault*) Function

This function retrieves or creates a data base entry given its key. The parameters have these interpretations.

team is a list representing a character string that uniquely identifies the database entry to be retrieved or created.

message is the address of a list known to the caller, which will be assigned a list representing an error message if any error occurs in the course of searching the database or creating a new entry.

fault is the address of an integer that will be set to a non-zero value if any error is caused by this function.

The pointer returned by this function is the address of the record whose key is given by the *team* parameter. If such a record is already in the database, its address is returned, but otherwise a new one is created whose address is then returned. The `reductions` field of a newly created entry will be zero.

In the course of searching the database, the `avm_compare` function is used, so the associated lists may be modified as noted in Section 3.1.5 [Comparison], page 80. It is not necessary for a client to include the header file '`compare.h`' or to call `avm_initialize_compare` in order to use the profile operations, because they are done automatically.

If an error message is assigned to the list referenced by *message*, the integer referenced by *fault* will be set to a non-zero value. The form of the error message will be a list in which each item is a list of character representations as per Appendix A [Character Table], page 123. It is the responsibility of the caller to dispose of the error message. Currently the only possible error is a memory overflow, which in this case is non-fatal.

`void` **avm_tally** (`char *`*filename*) Function

This function makes a table of the results stored in the data base built by the `avm_entries` function. The argument is the address of a null terminated character string containing the name of the file in which the results will be written. A file is opened and the table is written in a self explanatory text format, with columns labeled "reductions" and "invocations" among others. The latter contains the number of times the associated key was accessed through `avm_entries`.

The data written to the file should be taken with a grain of salt. It is computed using native integer and floating point arithmetic, with no checks made for overflow or roundoff error, and no guarantee of cross platform portability. The number of "reductions" means whatever the developer of the client program wants it to mean.

The following error messages are possible with this function, which will be written to standard error. None of them is fatal.

- *program-name*`: can't write` *filename*
- *program-name*`: can't write to` *filename*
- *program-name*`: can't close` *filename*
- *program-name*`: invalid profile identifier`

The last message is reported if any record in the database has a key that is not a list of valid character representations. The others are accompanied by an explanation from the standard `strerror` function if possible.

`void` **avm_initialize_profile** () Function

This function should be called before any of the other functions in this section in order to initialize the data base. Results are undefined if it is not called first.

void **avm_count_profile** () Function

This function can be called after the other functions in this section as a way of detecting memory leaks. If any storage remains unreclaimed that was created by the functions in this section, a warning message is written to standard error. If the `avm_count_lists` function is being used by the client program, it should be called after this one.

3.8 Emulation Primitives

The functions documented in this section can be used to take very specific control over the evaluation of virtual code applications. It is unlikely that a client program will have any need for them unless it aims to replace or extend the `avm_apply` function.

The virtual machine is somewhat removed from a conventional von Neumann model of computation, so emulating it in C or any other imperative language is less straightforward than one would prefer. An elaborate system of interdependent data structures is used to represent partially evaluated computations, which does not particularly lend itself to a convenient, modular API. The abstraction provided by the functions in this section is limited mainly to that of simple memory management and stack operations. Consequently, a developer wishing to build on them effectively would need to *grok* the data structures involved, which are described in some detail.

3.8.1 Lists of Pairs of Ports

A `port` is the name given to a type of pointer used in the library as the address of a place where a computational result yet to be evaluated will be sent. Ports are discussed further in Section 3.8.2 [Ports and Packets], page 106, but are mentioned here because it is sometimes necessary to employ a list of pairs of them. A pointer to such a list is declared as a `portal` type. It refers to a structure of the form

```
struct port_pair
{
  port left;
  port right;
  portal alters;
}
```

A small selection of functions for `portal` memory management is declared as follows in the header file '`portals.h`'. For reasons of C-ness, the type declarations themselves are forced to be in '`lists.h`'.

portal **avm_new_portal** (portal *alters*) Function

This function is used to create storage for a new `port_pair` structure, and returns a `portal` pointer to it if successful. If the storage can't be allocated, a `NULL` pointer is returned. The `alters` field of the result is initialized as the given parameter supplied by the caller. All other fields are filled with zeros.

void **avm_seal** (portal *fate*) Function

This function performs the reclamation of storage associated with `portal` pointers, either by freeing them or by consigning them temporarily to a local cache for performance reasons. Client programs should use only this function for disposing of `portal` storage rather than using `free` directly, so as to allow accurate record keeping.

`void` **avm_initialize_portals** () Function

This function should be called by a client program prior to calling either of the above memory management functions in order to initialize some local variables. Anomalous results are possible otherwise.

`void` **avm_count_portals** () Function

This function should be called at the end of a run or after the last call to any of the other functions in this section as a way of detecting memory leaks associated with `portal` pointers. A warning message will be written to standard error if any remains unreclaimed.

3.8.2 Ports and Packets

A pointer type declared as a `port` points to a structure in the following form, where a `flag` is an unsigned short integer type, and a `counter` is an unsigned long integer.

```
struct avm_packet
{
  port parent;
  counter errors;
  portal descendents;
  list impetus, contents;
  flag predicating;
};
```

For reasons that make sense to C, the `avm_packet` and `port` types are declared in `lists.h`, but a few memory management operations on them are available by way of functions declared in '`ports.h`'. The intended meaning of this structure is described presently, but first the memory management functions are as follows.

`port` **avm_newport** (`counter` *errors*, `port` *parent*, `int` *predicating*) Function

This function attempts to allocate storage for a new packet structure and returns its address if successful. If storage can not be allocated, a `NULL` pointer is returned. The `errors`, `parent`, and `predicating` fields are initialized with the parameters supplied by the caller. The rest of the structure is filled with zeros. A local memory cache is used for improved performance.

`void` **avm_sever** (`port` *appendage*) Function

This function reclaims the storage associated with a `port`, either freeing it entirely or holding it in a local cache. None of the entities that may be referenced by pointers within the structure are affected. Only this function should be used by client programs for disposing of ports, not the `free` function directly, or some internal bookkeeping will be disrupted. An internal error results if the argument is a `NULL` pointer.

`void` **avm_initialize_ports** () Function

This function must be called prior to calling either of the two above, in order to initialize some static variables.

void **avm_count_ports** () Function

This function may be called after the last call to any of the other functions in this section in order to detect and report unreclaimed storage associated with ports. A non-fatal warning will be written to standard error if any is detected, but otherwise there is no effect.

The interesting aspect of this data structure is the role it plays in capturing the state of a computation. For this purpose, it corresponds to a single node in a partially computed result to be represented by a `list` when it's finished. The nodes should be envisioned as a doubly-linked binary tree, except that the pair of `descendents` for each node is not yet known with certainty, so a list of alternatives must be maintained.

Because the computation is not completed while this data structure exists, there are always some empty fields in it. For example, the `descendents` and the `contents` fields embody the same information, the latter doing so in a compact as opposed to a more expanded form. Hence, it would be redundant for both fields to be non-empty at the same time. The data structure is built initially with `descendents` and no `contents`, only to be transformed into one with `contents` and no `descendents`.

The significance of each field in the structure can be summarized as follows.

`contents` If the computational result destined for the `port` pointing to this packet is not complete, then this field is `NULL` and the `descendents` are being computed. Otherwise, it contains the result of the computation.

`descendents`
This field points to a list of pairs of ports serving as the destinations for an ensemble of concurrent computations.[1] The `head` and `tail` of the `contents` are to be identified respectively with the `contents` of the `left` and `right port` in the first pair to finish being computed.

`parent` If this packet is addressed by the `left` or the `right` of `port` in one of the `descendents` of some other packet, then this field points to that packet.

`errors` A non-zero value in this field indicates that the result destined for the `contents` of this packet is expected to be an error message. If the exact level of error severity incurred in the computation of the `contents` matches this number, then the contents can be assigned the result, but otherwise the result should propagate to the `contents` of the `parent`.

`predicating`
A non-zero value in this field implies that the result destined for the `contents` of this packet is being computed in order to decide which arm of a conditional function should be chosen. I.e., a `NULL` result calls for the one that is invoked when the predicate is false.

`impetus` If the result destined for the `contents` of this packet is being computed in order to transform a virtual code fragment from its original form to an equivalent

[1] Earlier versions of `avram` included a bottom avoiding choice combinator that required this feature, but which has been withdrawn. A single pair of descendent ports would now suffice.

representation capable of being evaluated more directly, this field points to a `list` node at the root of the virtual code in its original form.

One of the hitherto undocumented fields in a `list` node structure declared in '`lists.h`' is called the `interpretation`, and is of type `list`. A client program delving into sufficient depth of detail to be concerned with ports and packets may reasonably assign the `interpretation` field of the `list` referenced by the `impetus` field in a packet to be a copy of the `contents` of the packet when they are eventually obtained. Doing so will save some time by eliminating the need for it to be recomputed if the same virtual code should be executed again.

If this course is taken, the `facilitator` field in a `list` node, also hitherto undocumented, should contain the address of the packet referring to the list node as its `impetus`. The reason for this additional link is so that it can be followed when the `impetus` of the packet is be cleared by `avm_dispose` in the event that the `list` node is freed before the computation completes. This action is performed in order to preclude a dangling pointer in the `impetus` field.

3.8.3 Instruction Stacks

A header file named '`instruct.h`' declares a number of memory management and stack operations on a data structure of the following form.

```
struct instruction_node
{
  port client;
  score sheet;
  struct avm_packet actor;
  struct avm_packet datum;
  instruction dependents;
};
```

In this structure, an `instruction` is a pointer to an `instruction_node`, a `score` is a pointer to a profile database entry as discussed in Section 3.7 [Profiling], page 103, and the `port` and `avm_packet` types are as described in Section 3.8.2 [Ports and Packets], page 106.

This data structure is appropriate for a simple virtual machine code evaluation strategy involving no concurrency. The strategy to evaluate an expression $f\ x$ would be based on a stack of these nodes threaded through the `dependents` field, and would proceed something like this.

1. The stack is initialized to contain a single node having f in its `actor.contents` field, and x in its `datum.contents` field.
2. The `client` in this node would refer to a static packet to whose `contents` field the final result will be delivered.
3. The evaluator examines the `actor.contents` field on the top of the stack, detects by its form the operation it represents, and decides whether it corresponds to one that can be evaluated immediately by way of a canned function available in the library. List reversal, transposition, and comparison would be examples of such operations.
4. If the operation can be performed in this way, the result is computed and assigned to the destination indicated by the `client` field.

5. If the operation is not easy enough to perform immediately but is of a form recognizable as a combination of simpler operations, it is decomposed into the simpler operations, and each of them is strategically positioned on the stack so as to effect the evaluation of the combination. For example, if f were of the form `compose(`g`,`h`)` (`silly` notation), the node with f and x would be popped, but a node with g as its `actor.contents` would be pushed, and then a node with h as its `actor.contents` and x as its `datum.contents` would be pushed. Furthermore, the `client` field of the latter node would point to the `datum.contents` of the one with g, and the `client` field of the one with g would point wherever the `client` of the popped node used to point.
6. If the operation indicated by the top `actor.contents` is neither implemented by a canned operation in the library nor easily decomposable into some that are, the evaluator can either give up or use virtual code to execute other virtual code. The latter trick is accomplished by pushing a node with f as its `datum.contents`, and a copy of a hard coded virtual code interpreter V as its `actor.contents`. The `client` of this node will point to the f in the original node so as to overwrite it when a simplified version is subsequently computed. The implementation of V is a straightforward exercise in `silly` programming.
7. In any case, the evaluator would continue working on the stack until everything on it has been popped, at which time the result of the entire computation will be found in the packet addressed by the `client` in the original instruction node.

What makes this strategy feasible to implement is the assumption of a sequential language, wherein synchronization incurs no cost and is automatic. The availability of any operand is implied by its position at the top of the stack. If you are reading this section with a view to implementing a concurrent or multi-threaded evaluation strategy, it will be apparent that further provisions would need to be made, such as that of a `data_ready` flag added to the `avm_packet` structure.

The following functions support the use of stacks of instruction nodes that would be needed in an evaluation strategy such as the one above.

`int` **avm_scheduled** (`list` *actor_contents*, `counter` *datum_errors*, `list` *datum_contents*, `port` *client*, `instruction` `*`*next*, `score` *sheet*) — Function

This function performs the memory allocation for instruction nodes. It attempts to create one and to initialize the fields with the given parameters, returning a pointer to it if successful. It returns a `NULL` pointer if the storage could not be allocated.

Copies of the `list` parameters `actor_contents` and `data_contents` are made by this function using `avm_copied`, so the originals still exist as far as the caller is concerned and will have to be deallocated separately from this structure. The copies are made only if the allocation succeeds.

Any fields other than those indicated by the parameters to this function are filled with zeros in the result.

`void` **avm_retire** (`instruction` `*`*done*) — Function

This function performs the storage reclamation of instructions, taking as its argument the instruction to be reclaimed. The `list` fields in the structure corresponding to the `list` parameters used when it was created are specifically reclaimed as well, using `avm_dispose`.

The argument to this function is the address of an `instruction` rather than just an `instruction` so that the `instruction` whose address is given may be reassigned as the `dependents` field of the deallocated node. In this way, the instructions can form a stack that is popped by this function.

This function cooperates with `avm_scheduled` in the use of a local cache of instruction nodes in the interest of better performance. Client modules should not attempt to allocate or reclaim instructions directly with `malloc` or `free`, but use only these functions.

It causes a fatal internal error to pass a `NULL` pointer to this function.

`void` **avm_reschedule** (`instruction *`*next*) Function

Given the address of an instruction pointer that may be regarded as the top of a stack of instructions threaded through the `dependents` field, this function exchanges the positions of the top instruction and the one below it. A fatal internal error is caused if there are fewer than two instructions in the stack.

A use for this function arises in the course of evaluating virtual code applications of the form `conditional(`p`,(`f`,`g`))` (in `silly` notation). The evaluation strategy would require pushing nodes for all three constituents, but with p pushed last (therefore evaluated first). The result of the evaluation of p would require either the top one or the one below it to be popped without being evaluated, depending on whether the result is empty.

`void` **avm_initialize_instruct** () Function

This function should be called before any of the instruction memory management functions is called in order to initialize some local data structures. Results are unpredictable without it.

`void` **avm_count_instruct** () Function

This function should be called after the last call to any of the other functions in this section in order to detect and report unreclaimed storage associated with them. A warning message will be written to standard error if any unreclaimed instructions remain. This function relies on the assumption that the memory management has been done only by way of the above functions.

3.9 External Library Maintenance

External mathematical library functions such as those documented in Appendix D [External Libraries], page 135 that are invoked from virtual code by the `library` combinator (Section 2.7.16.1 [Library combinator], page 59) are also accessible from C by way of a uniform API implemented by the functions declared in `libfuns.h`. This interface applies even to libraries implemented in Fortran such as `minpack`. This section briefly documents the functions in `libfuns.h` and sets out some recommeded guidelines for developers wishing to add support for other external libraries.

3.9.1 Calling existing library functions

Whatever data types a library function manipulates, its argument and its result are each ultimately encoded each by a single list as explained in Section 3.1.4 [Type Conversions], page 72. This representation allows all library functions to be invoked by a uniform calling convention as detailed below.

`list` **avm_library_call** (`list` *library_name*, `list` *function_name*, `list` *argument*, `int` **fault*) Function

This function serves as an interpreter of external library functions by taking a *library_name*, a *function_name*, and an *argument* to the result returned by the corresponding library function for the given *argument*.

The library and function names should be encoded as lists of character representations, the same as the arguments that would be used with the `library` combinator if it were being invoked by virtual code (with attention to the backward compatibility issue explained in Section 3.2 [Characters and Strings], page 84).

If an error occurs in the course of evaluating a library function, the integer referenced by *fault* will be assigned a non-zero value, and the result will be a list of character string representations explaining the error, such as `<'memory overflow'>`, for example. Otherwise, the list returned will encode the result of the library function in a way that depends on the particular function being evaluated.

`list` **avm_have_library_call** (`list` *library_name*, `list` *function_name*, `int` **fault*) Function

This function implements the `have` combinator described in Section 2.7.16.2 [Have combinator], page 60, which tests for the availability of a library function. The *library_name* and *function_name* parameters are as explained above for `avm_library_call`, and `fault` could signal an error similarly for this function as well.

The result returned will be an error message in the event of an error, or a list of pairs of strings otherwise. The list will be empty if the library function is not available. If the library function is available, the list will contain a single pair, as in

```
<(library_name,function_name)>
```

In addition, the list representation of the character string `'*'` can be specified as either the library name or the function name or both. This string is interpreted as a wild card and will cause all matching pairs of library and function names to be returned in the list.

`void` **avm_initialize_libfuns** () Function

This function initializes some static data structures used by the two functions above. It may be called optionally before the first call to either of them, but will be called automatically if not.

`void` **avm_count_libfuns** () Function

This function can be used as an aid to detecting memory leaks. It reclaims any data structures allocated by `avm_initialize_libfuns` and should be called towards the end of a run some time prior to `avm_count_lists` Section 3.1.1 [Simple Operations], page 66, if the latter is being used.

3.9.2 Implementing new library functions

Adding more external libraries to `avram` is currently a manual procedure requiring the attention of a developer conversant with C. To support a new library called `foobar`, these steps need to be followed at a minimum.

- Create a new file called '`foobar.h`' under the '`avm/`' directory in the main source tree whose name doesn't clash with any existing file names and preferably doesn't induce any proper prefixes among them. This file should contain at least these function declarations.

  ```
  extern list avm_foobar_call (list function_name,list argument,
                               int *fault);

  extern list avm_have_foobar_call (list function_name,int *fault);

  extern void avm_initialize_foobar ();

  extern void avm_count_foobar ();
  ```

  There should also be the usual preprocessor directives for '`include`' files. The naming convention shown should be followed in anticipation of automated support for these operations in the future.
- Add '`foobar.h`' to the list of other header files in '`avm/Makefile.am`'.
- Create a new file called '`foobar.c`' under the '`src/`' directory whose name doesn't clash with any existing file names to store most of the library interface code. It can start out with stubs for the functions declared in '`foobar.h`'.
- Add '`foobar.c`' to the list of other source files in '`src/Makefile.am`'
- Execute the following command in the main '`avram-x.x.x`' source directory where the file '`configure.in`' is found.

  ```
  aclocal \
  && automake --gnu --add-missing \
  && autoconf
  ```

  This command requires having `automake` and `autoconf` installed on your system.
- Make the following changes to '`libfuns.c`'.
  - Add the line `#include<avm/foobar.h>` after the other `include` directives.
  - Add the string `"foobar"` to the end of the array of `libnames` in `avm_initialize_libfuns`.
  - Add a call to `avm_initialize_foobar` to the body.
  - Add a call to `avm_count_foobar` to the body of `avm_count_libfuns`.
  - Add a case of the form

    ```
    case nn:
      return avm_foobar_call(function_name,argument,fault);
    ```

    after the last case in `avm_library_call`, being careful not to change the order, and using the same name as above in the file '`foobar.h`'.

  - Add a case of the form

    ```
    case nn:
      looked_up = avm_have_foobar_call(function_name,fault);
      break;
    ```

    after the last case in `avm_have_library_call`, being careful not to change the order, and using the same name as above in the file '`foobar.h`'.

- Edit '`foobar.c`' and '`foobar.h`' to suit, periodically compiling and testing by executing `make`.
- Package and install at will.

The functions shown above have the obvious interpretations, namely that `avm_foobar_call` evaluates a library function from the `foobar` library, and `avm_have_foobar_call` tests for a function's availability. The latter should interpret wild cards as explained in Section 3.9.1 [Calling existing library functions], page 111, but should return only a list of strings for the matching function names rather than a list of pairs of strings, as the library name is redundant. The remaining functions are for static initialization and reclamation.

These functions should consist mainly of boilerplate code similar to the corresponding functions in any of the other library source files, which should be consulted as examples. The real work would be done by other functions called by them. These should be statically declared within the '`.c`' source file and normally not listed in the '`.h`' header file unless there is some reason to think they may be of more general use. Any externally visible functions should have names beginning with `avm_` to avoid name clashes.

Some helpful hints are reported below for what they may be worth.

- The reason for doing this is to leverage off other people's intelligence, so generally `foobar.c` should contain only glue code for library routines developed elsewhere with great skill rather than reinventing them in some home grown way.
- The best numerical software is often written by Fortran programmers. Linking to a Fortran library is no problem on GNU systems provided that all variables are passed by reference and all arrays are converted to column order (Section 3.1.4 [Type Conversions], page 72).
- Most C++ programmers have yet to reach a comparable standard, but C++ libraries can also be linked by running `nm` on the static library file to find out the real names of the functions and `c++filt` to find out which is which. However, there is no obvious workaround for the use of so called derived classes by C++ programmers to simulate passing functions as parameters.
- Anything worth using can probably be found in the Debian archive.
- Not all libraries are sensible candidates for interfaces to `avram`. Typical design flaws are
  - irrepressible debugging messages written to `stderr` or `stdout` that are unfit for end user consumption
  - deliberately crashing the application if `malloc` fails
  - opaque data types with undocumented storage requirements
  - opaque data types that would be useful to store persistently but have platform specific binary representations

- heavily state dependent semantics
- identifiers with clashing names
- restrictive licenses

Some of these misfeatures have workarounds as explained next in Section 3.9.3 [Working around library misfeatures], page 114, at least if there's nothing else wrong with the library.

Those who support `avram` are always prepared to assist in the dissemination of worthwhile contributed library modules under terms compatible with [Copying], page 167, and under separate copyrights if preferred. Contributed modules can be integrated into the official source tree provided that they meet the following additional guidelines to those above.

- source code documentation and indentation according to GNU coding standards (`http://www.gnu.org/prep/standards`)
- sufficient stability for a semi-annual release cycle
- no run-time or compile-time dependence on any non-free software, although dynamic loading and client/server interaction are acceptable
- portable or at least unbreakable configuration by appropriate use of `autoconf` macros and conditional defines
- little or no state dependence at the level of the virtual code interface (i.e., pure functions or something like them, except for random number generators and related applications)
- adequate documentation for a section in Appendix D [External Libraries], page 135

3.9.3 Working around library misfeatures

As mentioned already (Section 3.9.2 [Implementing new library functions], page 112), some common problems with external libraries that are worthwhile in other respects are that they may generate unwelcome console output while running, they may follow ill defined memory management policies, and they may handle exceptions just by crashing themselves along with the client module.

An accumulation of techniques for coping with these issues (short of modifying the library source) has been collected into the API and made available by way of the header file '`mwrap.h`'. This section briefly documents how they might be put to use.

3.9.3.1 Inept excess verbiage

Although the author of a library function may take pride in putting its activities on display, it should be assumed that virtual code applications running on `avram` have other agendas for the console, so the library interface module should prevent direct output from the external library.

More thoughtful API's may have a verbosity setting, which should be used in preference to this workaround, but failing that, it is easy to dispense with console output generated by calls to external library functions by using some combination of the following functions.

void **avm_turn_off_stdout** () Function

Calling this function will suppress all output to the standard output stream until the next time `avm_turn_on_stdout` is called. Additional calls to this function without intervening calls to `avm_turn_on_stdout` may be made safely with no effect. The standard output stream is flushed as a side effect of calling this function.

void **avm_turn_on_stdout** () Function

Calling this function will allow output to the standard output stream to resume if it has been suppressed previously by a call to `avm_turn_off_stdout`. If `avm_turn_off_stdout` has not been previously called, this function has no effect. Any output that would have been sent to `stdout` during the time it was turned off will be lost.

void **avm_turn_off_stderr** () Function

This function performs a similar service to that of `avm_turn_off_stdout` but pertains to the standard error stream. The standard error and the standard output streams are controlled independently even if both of them are piped to the same console.

void **avm_turn_on_stderr** () Function

This function performs a similar service to that of `avm_turn_on_stdout` but pertains to the standard error stream.

As an example, the following code fragment will prevent any output to standard output taking place as a side effect of `blather`, but will allow error messages to standard error. Note that ouput should not be left permanently turned off.

```
...
#include <avm/mwrap.h>
...

x = y + z;
avm_turn_off_stdout ();
w = blather (foo, bar, baz);
avm_turn_on_stdout ();
return w;
...
```

One possible issue with these functions is that they rely on a feature of the GNU C library that might not be portable to non-GNU systems and has not been widely tested on other platforms.

Another issue is that a library function could be both careless enough to clutter the console unconditionally and meticulous enough to check for I/O errors after each attempted write. Writing while the output stream is disabled will return an I/O error to the caller (i.e., to the verbose library function) for appropriate action, which could include terminating the process.

3.9.3.2 Memory leaks

Incorrect memory management may undermine confidence in a library when one wonders what else it gets wrong, but if the worst it does is leave a few bytes unreclaimed, then help is at hand.

The first priority is to assess the seriousness of the situation. Similarly to the way library functions are bracketed with calls to those listed in Section 3.9.3.1 [Inept excess verbiage], page 114, the following functions are meant to be placed before and after a call to a library function either for diagnostic purposes or production use.

`void` **avm_manage_memory** () Function

> After this function is called, all subsequent calls to the standard C functions `malloc`, `free`, and `realloc` are intercepted and logged until the next time `avm_dont_manage_memory` is called. Furthermore, a complete record is maintained of the addresses and sizes of all allocated areas of memory during this time in a persistent data structure managed internally.

`void` **avm_dont_manage_memory** () Function

> Calling this function suspends the storage monitoring activities initiated by calling `avm_manage_memory`, but the record of allocated memory areas is not erased.

`void` **avm_debug_memory** () Function

> After this function is called and `avm_manage_memory` is also called, the standard output stream will display a running account of the sizes and addresses of all memory allocations or deallocations as they occur until the next call to either `avm_dont_debug_memory` or `avm_dont_manage_memory`.

`void` **avm_dont_debug_memory** () Function

> This function stops the output being sent to `stdout` caused by `avm_debug_memory`, if any, but has no effect on the logging of memory management events preformed due to `avm_manage_memory`.

While the latter two are not useful in production code, they can help to clarify an inadequately documented API during development by experimentally identifying the functions that cause memory to be allocated. They can also provide the answer to questions like whether separate copies are made from arrays passed to functions (useful for knowing when it's appropriate to free them).

Although the console output reveals everything there is to know about memory management during the selected window, the question of unreclaimed storage is more directly settled by the following functions.

`void` **avm_initialize_mwrap** () Function

> This function has to be called before any other functions from '`mwrap.h`' in order to clean the slate and prepare the static data structures for use. This function might not have to be called explicitly if the client module is part of `avram`, whose main program would have already called it. There is no harm in calling it repeatedly.

`void` **avm_count_mwrap** () Function

> This function should be called after the last call to any other functions in '`mwrap.h`', when it is expected that all storage that was allocated while `avm_manage_memory` was in effect should have been reclaimed.

If there is no unreclaimed storage allocated during an interval when memory was being managed, this function returns uneventfully. However, if any storage remains unreclaimed, a message stating the number of bytes is written to `stderr`.

If `avm_debug_memory` is also in effect when this function detects unreclaimed storage, an itemized list of the unreclaimed memory addresses and their sizes is written to standard output.

Of course, in order for `avm_count_mwrap` to report meaningful results, any memory that is allocated during the interval between calls to `avm_manage_memory` and `avm_dont_manage_memory` must have been given an opportunity to be reclaimed also while this logging mechanism is in effect. However, there may be arbitrarily many intervening intervals during which it is suspended.

On the other hand, any storage that is allocated when memory is not being managed must not be freed at a time when it is (except for freeing a `NULL` pointer, which is tolerated but not encouraged). Doing so raises an internal error, causing termination with extreme prejudice. This behavior is a precaution against library functions freeing storage that they didn't allocate, which would mean no memory is safe and it's better for `avram` not to continue.

If these investigations uncover no evidence of a memory leak, then perhaps the relevant library functions are reliable enough to run without supervisory memory management. Alternatively, when memory leaks are indicated, the next function provides a simple remedy.

`void` **avm_free_managed_memory** () Function

This function causes all storage to be reclaimed that was allocated at any time while logging of memory allocation was in effect (i.e., whenever `avm_manage_memory` had been called more recently than `avm_dont_manage_memory`). When the storage is freed, no further record of it is maintained.

A side effect of this function is to call `avm_dont_manage_memory` and therefore leave memory management turned off.

This last function when used in conjunction with the others is therefore the workaround for library functions that don't clean up after themselves. It may be important to do it for them if repeated calls to the library function are expected, which would otherwise cause unreclaimed storage to accumulate until it curtailed other operations.

One small issue with this function is the assumption that unreclaimed storage is really a leak and not internal library data that is designed to persist between calls. If this assumption is not valid, breakage will occur. However, libraries deliberately making use of persistent data are likely to have initialization and destructor functions as part of their API's, so this assumption is often justified if they don't.

An example of using these functions is given below.

In this example, `allocated_library_object` is a hypothetical function exported by an external library that causes storage to be allocated, and `library_reclamation_routine` is provided by the same library ostensibly to reclaim the storage thus allocated. However, the latter is suspected of memory leaks.

The variable `my_data` is declared and used by an `avram` developer who is presumably competent to reclaim it correctly, rather than it being part of an external library. Memory management is therefore enabled during the calls to the library routines but not at other times.

The call to `avm_count_mwrap` is redundant immediately after a call to `avm_free_managed_memory`, because with all managed memory having been freed, no memory leak will ever be detected, but it is included for illustrative purposes.

```
#include <avm/mwrap.h>
...

{
  void *behemoth;
  char *my_data;

  avm_initialize_mwrap ();
  avm_manage_memory ();
  behemoth = allocated_library_object (foo, bar);
  avm_dont_manage_memory ();
  my_data = (char *) malloc (100);
  ...
  free (my_data);
  avm_manage_memory ();
  library_reclamation_routine (&behemoth);
  avm_free_managed_memory ();
  avm_count_mwrap ();
  return;
}
```

It might be a cleaner solution in some sense to omit the call to `library_reclamation_routine` entirely, because the storage allocated during the call to `allocated_library_object` will be reclaimed perfectly well by `avm_free_managed_memory` without it. Doing so may also be the only option if the library reclamation routine is either extremely unreliable or non-existent. However, the style above is to be preferred for portability if possible. The memory management functions rely on the availability of the system header file `malloc.h`, and GNU C library features whose portability is not assured. If the required features are not detected on the host system at configuration time, conditional directives in the `avram` source will make the `avm_*` memory management functions perform no operations, and the responsibility for memory management will devolve to the possibly less robust external library implementation.

3.9.3.3 Suicidal exception handling

An inconvenient characteristic of some external library functions is to terminate the program rather than returning an error status to the caller for routine events such as a failure of memory allocation. Although in many cases there is no simple workaround for this behavior, memory allocation failures at least can be detected and preventive action taken by using the functions described in this section.

The general approach is to use memory management functions from '`mwrap.h`' as described previously (Section 3.9.3.2 [Memory leaks], page 115), while additionally registering a return destination for a non-local jump to be taken in the event of a memory overflow.

The jump is taken when an external library function calls `malloc` or `realloc` unsuccessfully. The jump avoids passing control back to the library function, thereby denying it the opportunity to abort, but restores the context to that of the jump destination almost as if the library function and all of its intervening callers had returned normally.

The interface is similar to that of the standard `setjmp` function defined in the system header file `setjmp.h`, and in fact is built on it, but differs in that the client module does not explicitly refer to jump buffers. Instead, the `mwrap` module internally maintains a stack of return destinations.

If a jump is taken, it always goes to the most recently registered destination. It may revert to the previously registered destination only when the current one is cleared. This organization provides the necessary flexibility for multiple clients and recursion, but it necessitates a protocol whereby each registration of a destination must be explicitly cleared exactly once.

The following functions implement these two features.

`int` **avm_setjmp** () — Function

> This function specifies the point to which control will pass by a non-local jump if there is insufficient memory to complete a subsequent `malloc` or `realloc` operation. Only the operations that take place while memory is being managed due to `avm_manage_memory` are affected (Section 3.9.3.2 [Memory leaks], page 115).
>
> The function returns zero when it is called normally and successfully registers the return point.
>
> It returns a non-zero value when it has been entered by a non-local jump (i.e., when `malloc` or `realloc` has reported insufficient memory while memory management is active), or when the return point could not be successfully registered due to insufficient memory. The client need not distinguish between these two cases, because both correspond to memory overflows and the destination must be cleared by `avm_clearjmp` regardless.
>
> When a non-zero value is returned due to this function being reached by a non-local jump, it has the side effects of reclaiming all managed memory by calling `avm_free_managed_memory` and disabling memory management by calling `avm_dont_manage_memory`.

`void` **avm_clearjmp** () — Function

> This function cancels the effect of `avm_setjmp ()` by preventing further non-local jumps to its destination if the destination was successfully registered, or by acknowledging unsuccessful registration otherwise. It should be called before exiting any function that calls `avm_setjmp ()` or anomalous results may ensue.

The memory management functions `avm_manage_memory` and `avm_dont_manage_memory` can be useful with or without `avm_setjmp`, depending on how much of a workaround is needed for a given library. If a library does not abort on memory overflows, there is no need to use `avm_setjmp`, while it may still be appropriate to use the other functions against memory leaks.

Calling avm_clearjmp is particularly important if a client module with memory management that doesn't use avm_setjmp is invoked subsequently to one that does, so that memory overflows in the latter won't cause an attempted jump to a stale destination.

A further complication that arises from careful consideration of these issues is the situation of a client module that does not intend to use avm_setjmp but is called (perhaps indirectly) by one that does. The latter will have registered a return destination that remains active and valid even if the former refrains from doing so, thereby allowing a branch to be taken that should have been prevented. Although it is an unusual situation, it can be accommodated by the following function.

void **avm_setnonjump** () Function

This function temporarily inhibits non-local jumps to destinations previously registered by avm_setjmp until the next time avm_clearjmp is called. Thereafter, any previously registered destinations are reinstated.

A sketch of how some of these functions might be used to cope with library functions that would otherwise terminate the program in the event of a memory overflow is shown below. The GNU libc reference manual contains a related discussion of non-local jumps.

```
#include <avm/mwrap.h>
...

int
function foobar (foo, bar)
...
{
char *my_data;

  my_data = (char *) malloc (100);
  if (avm_setjmp () != 0)
    {
      avm_clearjmp ();
      avm_turn_on_stdout ();       /* reaching here */
      free (my_data);              /* means malloc  */
      return ABNORMAL_STATUS;      /* failed below  */
    }
  avm_turn_off_stdout ();
  avm_manage_memory ();
  ...
  call_library_functions (foo, bar);    /* may jump */
  ...                                   /* to above */
  avm_free_managed_memory ();
  avm_turn_on_stdout ();
  avm_clearjmp ();
  free (my_data);             /* reaching here means */
  return OK_STATUS;           /* jumping wasn't done */
}
```

Portability issues with these functions are not well known at this writing. If the configuration script for avram fails to detect the required features in setjmp.h on the host system, conditional compilation directives will disable the functions avm_setjmp, avm_clearjmp, and avm_setnonjmp. However, it may still be possible for the other avm_* memory management functions to be configured.

If setjmp is not configured, the avm_setjmp function is still callable but will always return a value of zero, and will provide no protection against external library functions aborting the program. The other two will perform no operation and return.

Appendix A Character Table

This table lists the representations used by `avram` for characters. The left column shows the character code in decimal. For printable characters, the middle column shows the character. The right column shows the representation used. For example, the letter `A` has character code 65, and the representation `(nil,(((nil,(nil,(nil,nil))),nil),(nil,nil)))`.

These representations were generated automatically to meet various helpful criteria, and are not expected to change in future releases. No character representation coincides with the representations used for boolean values, natural numbers, character strings, pairs of characters, or certain other data types beyond the scope of this document. An easy algorithm for lexical sorting is possible. Subject to these criteria, the smallest possible trees were chosen.

```
 0   (nil,(nil,(nil,((nil,nil),(nil,nil)))))
 1   (nil,(nil,((nil,nil),(nil,nil))))
 2   (nil,(nil,((nil,nil),(nil,(nil,nil)))))
 3   (nil,(nil,((nil,(nil,nil)),(nil,nil))))
 4   (nil,(nil,(((nil,nil),nil),(nil,nil))))
 5   (nil,(nil,(((nil,nil),(nil,nil)),nil)))
 6   (nil,(nil,((((nil,nil),(nil,nil)),nil),nil)))
 7   (nil,((nil,nil),(nil,nil)))
 8   (nil,((nil,nil),(nil,(nil,nil))))
 9   (nil,((nil,nil),(nil,(nil,(nil,nil)))))
10   (nil,((nil,nil),(nil,(nil,(nil,(nil,nil))))))
11   (nil,((nil,nil),(nil,((nil,nil),(nil,nil)))))
12   (nil,((nil,nil),(nil,((nil,(nil,nil)),nil))))
13   (nil,((nil,nil),(nil,(((nil,nil),nil),nil))))
14   (nil,((nil,nil),((nil,nil),(nil,nil))))
15   (nil,((nil,nil),((nil,nil),(nil,(nil,nil)))))
16   (nil,((nil,nil),((nil,(nil,nil)),nil)))
17   (nil,((nil,nil),((nil,(nil,nil)),(nil,nil))))
18   (nil,((nil,nil),((nil,(nil,(nil,nil))),nil)))
19   (nil,((nil,nil),(((nil,nil),nil),(nil,nil))))
20   (nil,((nil,nil),(((nil,nil),(nil,nil)),nil)))
21   (nil,((nil,(nil,nil)),(nil,nil)))
22   (nil,((nil,(nil,nil)),(nil,(nil,nil))))
23   (nil,((nil,(nil,nil)),(nil,(nil,(nil,nil)))))
24   (nil,((nil,(nil,nil)),(nil,((nil,nil),nil))))
25   (nil,((nil,(nil,nil)),((nil,nil),nil)))
26   (nil,((nil,(nil,nil)),((nil,nil),(nil,nil))))
27   (nil,((nil,(nil,nil)),((nil,(nil,nil)),nil)))
28   (nil,((nil,(nil,nil)),(((nil,nil),nil),nil)))
29   (nil,((nil,(nil,(nil,nil))),(nil,nil)))
30   (nil,((nil,(nil,(nil,nil))),(nil,(nil,nil))))
31   (nil,((nil,(nil,(nil,nil))),((nil,nil),nil)))
32   (nil,((nil,(nil,(nil,(nil,nil)))),(nil,nil)))
33 ! (nil,((nil,(nil,((nil,nil),nil))),(nil,nil)))
34 " (nil,((nil,(nil,((nil,nil),(nil,nil)))),nil))
```

```
35 # (nil,((nil,((nil,nil),nil)),(nil,nil)))
36 $ (nil,((nil,((nil,nil),nil)),(nil,(nil,nil))))
37 % (nil,((nil,((nil,nil),(nil,nil))),nil))
38 & (nil,((nil,((nil,nil),(nil,nil))),(nil,nil)))
39 ' (nil,((nil,((nil,nil),(nil,(nil,nil)))),nil))
40 ( (nil,((nil,((nil,(nil,nil)),nil)),(nil,nil)))
41 ) (nil,((nil,((nil,(nil,nil)),(nil,nil))),nil))
42 * (nil,((nil,(((nil,nil),nil),nil)),(nil,nil)))
43 + (nil,((nil,(((nil,nil),nil),(nil,nil))),nil))
44 , (nil,((nil,(((nil,nil),(nil,nil)),nil)),nil))
45 - (nil,(((nil,nil),nil),(nil,nil)))
46 . (nil,(((nil,nil),nil),(nil,(nil,nil))))
47 / (nil,(((nil,nil),nil),(nil,(nil,(nil,nil)))))
48 0 (nil,(((nil,nil),nil),((nil,nil),(nil,nil))))
49 1 (nil,(((nil,nil),nil),((nil,(nil,nil)),nil)))
50 2 (nil,(((nil,nil),(nil,nil)),nil))
51 3 (nil,(((nil,nil),(nil,nil)),(nil,nil)))
52 4 (nil,(((nil,nil),(nil,nil)),(nil,(nil,nil))))
53 5 (nil,(((nil,nil),(nil,nil)),((nil,nil),nil)))
54 6 (nil,(((nil,nil),(nil,(nil,nil))),nil))
55 7 (nil,(((nil,nil),(nil,(nil,nil))),(nil,nil)))
56 8 (nil,(((nil,nil),(nil,(nil,(nil,nil)))),nil))
57 9 (nil,(((nil,nil),((nil,nil),nil)),(nil,nil)))
58 : (nil,(((nil,nil),((nil,nil),(nil,nil))),nil))
59 ; (nil,(((nil,nil),((nil,(nil,nil)),nil)),nil))
60 < (nil,(((nil,(nil,nil)),nil),(nil,nil)))
61 = (nil,(((nil,(nil,nil)),nil),(nil,(nil,nil))))
62 > (nil,(((nil,(nil,nil)),(nil,nil)),nil))
63 ? (nil,(((nil,(nil,nil)),(nil,nil)),(nil,nil)))
64 @ (nil,(((nil,(nil,nil)),(nil,(nil,nil))),nil))
65 A (nil,(((nil,(nil,(nil,nil))),nil),(nil,nil)))
66 B (nil,(((nil,(nil,(nil,nil))),(nil,nil)),nil))
67 C (nil,(((nil,((nil,nil),nil)),nil),(nil,nil)))
68 D (nil,(((nil,((nil,nil),nil)),(nil,nil)),nil))
69 E (nil,((((nil,nil),nil),nil),(nil,nil)))
70 F (nil,((((nil,nil),nil),nil),(nil,(nil,nil))))
71 G (nil,((((nil,nil),nil),(nil,nil)),nil))
72 H (nil,((((nil,nil),nil),(nil,nil)),(nil,nil)))
73 I (nil,((((nil,nil),nil),(nil,(nil,nil))),nil))
74 J (nil,((((nil,nil),(nil,nil)),nil),(nil,nil)))
75 K (nil,((((nil,nil),(nil,nil)),(nil,nil)),nil))
76 L (nil,((((nil,(nil,nil)),nil),nil),(nil,nil)))
77 M (nil,((((nil,(nil,nil)),nil),(nil,nil)),nil))
78 N (nil,(((((nil,nil),nil),nil),nil),(nil,nil)))
79 O (nil,(((((nil,nil),nil),nil),(nil,nil)),nil))
80 P ((nil,nil),(nil,nil))
81 Q ((nil,nil),(nil,(nil,nil)))
82 R ((nil,nil),(nil,(nil,(nil,nil))))
83 S ((nil,nil),(nil,(nil,(nil,(nil,nil)))))
```

```
 84 T ((nil,nil),(nil,(nil,(nil,(nil,(nil,nil))))))
 85 U ((nil,nil),(nil,(nil,((nil,(nil,nil)),nil))))
 86 V ((nil,nil),(nil,(nil,(((nil,nil),nil),nil))))
 87 W ((nil,nil),(nil,((nil,nil),(nil,nil))))
 88 X ((nil,nil),(nil,((nil,(nil,nil)),nil)))
 89 Y ((nil,nil),(nil,((nil,(nil,nil)),(nil,nil))))
 90 Z ((nil,nil),(nil,((nil,(nil,(nil,nil))),nil)))
 91 [ ((nil,nil),(nil,((nil,((nil,nil),nil)),nil)))
 92 \ ((nil,nil),(nil,(((nil,nil),nil),nil)))
 93 ] ((nil,nil),(nil,(((nil,nil),nil),(nil,nil))))
 94 ^ ((nil,nil),(nil,(((nil,nil),(nil,nil)),nil)))
 95 _ ((nil,nil),(nil,(((nil,(nil,nil)),nil),nil)))
 96 ` ((nil,nil),(nil,((((nil,nil),nil),nil),nil)))
 97 a ((nil,nil),((nil,nil),(nil,nil)))
 98 b ((nil,nil),((nil,nil),(nil,(nil,nil))))
 99 c ((nil,nil),((nil,nil),(nil,(nil,(nil,nil)))))
100 d ((nil,nil),((nil,nil),((nil,nil),(nil,nil))))
101 e ((nil,nil),((nil,nil),((nil,(nil,nil)),nil)))
102 f ((nil,nil),((nil,(nil,nil)),nil))
103 g ((nil,nil),((nil,(nil,nil)),(nil,nil)))
104 h ((nil,nil),((nil,(nil,nil)),(nil,(nil,nil))))
105 i ((nil,nil),((nil,(nil,nil)),((nil,nil),nil)))
106 j ((nil,nil),((nil,(nil,(nil,nil))),nil))
107 k ((nil,nil),((nil,(nil,(nil,nil))),(nil,nil)))
108 l ((nil,nil),((nil,(nil,(nil,(nil,nil)))),nil))
109 m ((nil,nil),((nil,((nil,nil),nil)),(nil,nil)))
110 n ((nil,nil),((nil,((nil,nil),(nil,nil))),nil))
111 o ((nil,nil),((nil,((nil,(nil,nil)),nil)),nil))
112 p ((nil,nil),(((nil,nil),nil),(nil,nil)))
113 q ((nil,nil),(((nil,nil),nil),(nil,(nil,nil))))
114 r ((nil,nil),(((nil,nil),(nil,nil)),nil))
115 s ((nil,nil),(((nil,nil),(nil,nil)),(nil,nil)))
116 t ((nil,nil),(((nil,nil),(nil,(nil,nil))),nil))
117 u ((nil,nil),(((nil,(nil,nil)),nil),(nil,nil)))
118 v ((nil,nil),(((nil,(nil,nil)),(nil,nil)),nil))
119 w ((nil,nil),((((nil,nil),nil),nil),(nil,nil)))
120 x ((nil,nil),((((nil,nil),nil),(nil,nil)),nil))
121 y ((nil,nil),(((((nil,nil),nil),nil),nil),nil))
122 z ((nil,(nil,nil)),(nil,nil))
123 { ((nil,(nil,nil)),(nil,(nil,(nil,nil))))
124 | ((nil,(nil,nil)),(nil,(nil,(nil,(nil,nil)))))
125 } ((nil,(nil,nil)),(nil,((nil,nil),nil)))
126 ~ ((nil,(nil,nil)),(nil,((nil,nil),(nil,nil))))
127   ((nil,(nil,nil)),(nil,((nil,(nil,nil)),nil)))
128   ((nil,(nil,nil)),((nil,nil),(nil,nil)))
129   ((nil,(nil,nil)),((nil,nil),(nil,(nil,nil))))
130   ((nil,(nil,nil)),((nil,(nil,nil)),nil))
131   ((nil,(nil,nil)),((nil,(nil,nil)),(nil,nil)))
132   ((nil,(nil,nil)),((nil,(nil,(nil,nil))),nil))
```

```
133   ((nil,(nil,nil)),(((nil,nil),nil),(nil,nil)))
134   ((nil,(nil,nil)),(((nil,nil),(nil,nil)),nil))
135   ((nil,(nil,(nil,nil))),(nil,nil))
136   ((nil,(nil,(nil,nil))),(nil,(nil,nil)))
137   ((nil,(nil,(nil,nil))),(nil,(nil,(nil,nil))))
138   ((nil,(nil,(nil,nil))),(nil,((nil,nil),nil)))
139   ((nil,(nil,(nil,nil))),((nil,nil),(nil,nil)))
140   ((nil,(nil,(nil,nil))),((nil,(nil,nil)),nil))
141   ((nil,(nil,(nil,(nil,nil)))),(nil,nil))
142   ((nil,(nil,(nil,(nil,nil)))),(nil,(nil,nil)))
143   ((nil,(nil,(nil,(nil,nil)))),((nil,nil),nil))
144   ((nil,(nil,(nil,(nil,(nil,nil))))),(nil,nil))
145   ((nil,(nil,(nil,((nil,nil),nil)))),(nil,nil))
146   ((nil,(nil,((nil,nil),nil))),(nil,nil))
147   ((nil,(nil,((nil,nil),(nil,nil)))),(nil,nil))
148   ((nil,(nil,((nil,(nil,nil)),nil))),(nil,nil))
149   ((nil,(nil,(((nil,nil),nil),nil))),(nil,nil))
150   ((nil,((nil,nil),nil)),(nil,nil))
151   ((nil,((nil,nil),nil)),(nil,(nil,nil)))
152   ((nil,((nil,nil),nil)),(nil,(nil,(nil,nil))))
153   ((nil,((nil,nil),nil)),(nil,((nil,nil),nil)))
154   ((nil,((nil,nil),nil)),((nil,nil),(nil,nil)))
155   ((nil,((nil,nil),nil)),((nil,(nil,nil)),nil))
156   ((nil,((nil,nil),(nil,nil))),(nil,nil))
157   ((nil,((nil,nil),(nil,nil))),(nil,(nil,nil)))
158   ((nil,((nil,nil),(nil,(nil,nil)))),(nil,nil))
159   ((nil,((nil,nil),((nil,nil),nil))),(nil,nil))
160   ((nil,((nil,(nil,nil)),nil)),(nil,nil))
161   ((nil,((nil,(nil,nil)),nil)),(nil,(nil,nil)))
162   ((nil,((nil,(nil,nil)),nil)),((nil,nil),nil))
163   ((nil,((nil,(nil,nil)),(nil,nil))),(nil,nil))
164   ((nil,((nil,(nil,(nil,nil))),nil)),(nil,nil))
165   ((nil,((nil,((nil,nil),nil)),nil)),(nil,nil))
166   ((nil,(((nil,nil),nil),nil)),(nil,nil))
167   ((nil,(((nil,nil),nil),(nil,nil))),(nil,nil))
168   ((nil,(((nil,nil),(nil,nil)),nil)),(nil,nil))
169   ((nil,(((nil,(nil,nil)),nil),nil)),(nil,nil))
170   ((nil,((((nil,nil),nil),nil),nil)),(nil,nil))
171   (((nil,nil),nil),(nil,nil))
172   (((nil,nil),nil),(nil,(nil,nil)))
173   (((nil,nil),nil),(nil,(nil,(nil,nil))))
174   (((nil,nil),nil),(nil,(nil,(nil,(nil,nil)))))
175   (((nil,nil),nil),(nil,(nil,((nil,nil),nil))))
176   (((nil,nil),nil),(nil,((nil,nil),nil)))
177   (((nil,nil),nil),(nil,((nil,nil),(nil,nil))))
178   (((nil,nil),nil),(nil,((nil,(nil,nil)),nil)))
179   (((nil,nil),nil),(nil,(((nil,nil),nil),nil)))
180   (((nil,nil),nil),((nil,nil),(nil,nil)))
181   (((nil,nil),nil),((nil,nil),(nil,(nil,nil))))
```

182	(((nil,nil),nil),((nil,(nil,nil)),nil))
183	(((nil,nil),nil),((nil,(nil,nil)),(nil,nil)))
184	(((nil,nil),nil),((nil,(nil,(nil,nil))),nil))
185	(((nil,nil),nil),(((nil,nil),nil),(nil,nil)))
186	(((nil,nil),nil),(((nil,nil),(nil,nil)),nil))
187	(((nil,nil),(nil,nil)),(nil,nil))
188	(((nil,nil),(nil,nil)),(nil,(nil,nil)))
189	(((nil,nil),(nil,nil)),(nil,(nil,(nil,nil))))
190	(((nil,nil),(nil,nil)),(nil,((nil,nil),nil)))
191	(((nil,nil),(nil,nil)),((nil,(nil,nil)),nil))
192	(((nil,nil),(nil,(nil,nil))),(nil,nil))
193	(((nil,nil),(nil,(nil,nil))),(nil,(nil,nil)))
194	(((nil,nil),(nil,(nil,(nil,nil)))),(nil,nil))
195	(((nil,nil),(nil,((nil,nil),nil))),(nil,nil))
196	(((nil,nil),((nil,nil),nil)),(nil,nil))
197	(((nil,nil),((nil,nil),nil)),(nil,(nil,nil)))
198	(((nil,nil),((nil,nil),(nil,nil))),(nil,nil))
199	(((nil,nil),((nil,(nil,nil)),nil)),(nil,nil))
200	(((nil,nil),(((nil,nil),nil),nil)),(nil,nil))
201	(((nil,(nil,nil)),nil),(nil,nil))
202	(((nil,(nil,nil)),nil),(nil,(nil,nil)))
203	(((nil,(nil,nil)),nil),(nil,(nil,(nil,nil))))
204	(((nil,(nil,nil)),nil),(nil,((nil,nil),nil)))
205	(((nil,(nil,nil)),nil),((nil,nil),(nil,nil)))
206	(((nil,(nil,nil)),nil),((nil,(nil,nil)),nil))
207	(((nil,(nil,nil)),(nil,nil)),(nil,nil))
208	(((nil,(nil,nil)),(nil,nil)),(nil,(nil,nil)))
209	(((nil,(nil,nil)),(nil,(nil,nil))),(nil,nil))
210	(((nil,(nil,nil)),((nil,nil),nil)),(nil,nil))
211	(((nil,(nil,(nil,nil))),nil),(nil,nil))
212	(((nil,(nil,(nil,nil))),nil),(nil,(nil,nil)))
213	(((nil,(nil,(nil,nil))),nil),((nil,nil),nil))
214	(((nil,(nil,(nil,nil))),(nil,nil)),(nil,nil))
215	(((nil,(nil,(nil,(nil,nil)))),nil),(nil,nil))
216	(((nil,(nil,((nil,nil),nil))),nil),(nil,nil))
217	(((nil,((nil,nil),nil)),nil),(nil,nil))
218	(((nil,((nil,nil),nil)),nil),(nil,(nil,nil)))
219	(((nil,((nil,nil),nil)),nil),((nil,nil),nil))
220	(((nil,((nil,nil),nil)),(nil,nil)),(nil,nil))
221	(((nil,((nil,nil),(nil,nil))),nil),(nil,nil))
222	(((nil,((nil,(nil,nil)),nil)),nil),(nil,nil))
223	(((nil,(((nil,nil),nil),nil)),nil),(nil,nil))
224	((((nil,nil),nil),nil),(nil,nil))
225	((((nil,nil),nil),nil),(nil,(nil,nil)))
226	((((nil,nil),nil),nil),(nil,(nil,(nil,nil))))
227	((((nil,nil),nil),nil),(nil,((nil,nil),nil)))
228	((((nil,nil),nil),nil),((nil,nil),nil))
229	((((nil,nil),nil),nil),((nil,nil),(nil,nil)))
230	((((nil,nil),nil),nil),((nil,(nil,nil)),nil))

```
231   ((((nil,nil),nil),nil),(((nil,nil),nil),nil))
232   ((((nil,nil),nil),(nil,nil)),(nil,nil))
233   ((((nil,nil),nil),(nil,nil)),(nil,(nil,nil)))
234   ((((nil,nil),nil),(nil,(nil,nil))),(nil,nil))
235   ((((nil,nil),nil),((nil,nil),nil)),(nil,nil))
236   ((((nil,nil),(nil,nil)),nil),(nil,nil))
237   ((((nil,nil),(nil,nil)),nil),(nil,(nil,nil)))
238   ((((nil,nil),(nil,nil)),(nil,nil)),(nil,nil))
239   ((((nil,nil),(nil,(nil,nil))),nil),(nil,nil))
240   ((((nil,nil),((nil,nil),nil)),nil),(nil,nil))
241   ((((nil,(nil,nil)),nil),nil),(nil,nil))
242   ((((nil,(nil,nil)),nil),nil),(nil,(nil,nil)))
243   ((((nil,(nil,nil)),nil),nil),((nil,nil),nil))
244   ((((nil,(nil,nil)),nil),(nil,nil)),(nil,nil))
245   ((((nil,(nil,nil)),(nil,nil)),nil),(nil,nil))
246   ((((nil,(nil,(nil,nil))),nil),nil),(nil,nil))
247   ((((nil,((nil,nil),nil)),nil),nil),(nil,nil))
248   (((((nil,nil),nil),nil),nil),(nil,nil))
249   (((((nil,nil),nil),nil),nil),(nil,(nil,nil)))
250   (((((nil,nil),nil),nil),nil),((nil,nil),nil))
251   (((((nil,nil),nil),nil),(nil,nil)),(nil,nil))
252   (((((nil,nil),nil),(nil,nil)),nil),(nil,nil))
253   (((((nil,nil),(nil,nil)),nil),nil),(nil,nil))
254   (((((nil,(nil,nil)),nil),nil),nil),(nil,nil))
255   ((((((nil,nil),nil),nil),nil),nil),(nil,nil))
```

Appendix B Reference Implementations

This appendix contains some `silly` source code for several functions that are mentioned in Section 2.7 [Virtual Code Semantics], page 33, for specifying the virtual machine code semantics, namely `pairwise`, `transition`, `insert` and `replace`.

The intention is to specify the virtual machine mathematically with a minimum of hand waving, by using only simple equations and small fragments of `silly` code, which has a straightforward semantics. However, the `silly` code fragments are more significant in some cases than what could fit into a few lines or be mechanically derived from an equation.

The purpose of this appendix is therefore to avoid leaving any gaps in the construction by demonstrating that everything mentioned can be done. None of this code is needed for any practical purpose, because its functionality is inherent in the virtual machine, but it shows how certain operations would be specified if they were not built in.

B.1 Pairwise

This `silly` code fragment is mentioned in Section 2.7.13.3 [Reduce], page 48, in the discussion of `reduce`, and is provided as an example of a solution to equations *E1* to *E3*. It is written in the style of a higher order function, in that it takes a function f as an argument and returns another function, [[`pairwise`]] f as a result.

```
self     = left
argument = right
head     = left
tail     = right

pairwise =

compose(
   refer,
   compose(
      bu(
         conditional,
         conditional(argument,compose(tail,argument),constant nil)),
      couple(
         (hired couple)(
            (hired compose)(
               identity,
               constant (hired fan head)(
                  argument,
                  compose(tail,argument))),
            constant (hired meta)(
               self,
               compose(tail,compose(tail,argument)))),
         constant argument)))
```

To see how this works, one should evaluate it symbolically with an unknown f, which will result in some `silly` pseudocode, and then evaluate that symbolically with some sample lists.

B.2 Insert

This function is mentioned in Section 2.7.13.4 [Sort], page 49, on sorting. It takes the virtual code for a partial order relational operator and returns the code for a function of two arguments. The left argument is a list item and the right argument is a list of items of the same type, which is already sorted with respect to the relational operator given as the argument to `insert`. The result of the function returned by `insert` is a list similar to its right argument but with the left argument inserted in the proper position to maintain the order.

This code makes use of the `self`, `argument`, `head` and `tail` declarations associated with `pairwise`.

```
insert =

bu(compose,refer) (hired conditional)(
   constant compose(right,argument),
   couple(
      (hired conditional)(
         (hired compose)(
            identity,
            constant compose(
               couple(left,compose(head,right)),
               argument)),
         constant (
            argument,
            couple(
               compose(head,compose(right,argument)),
               (hired meta)(
                  self,
                  couple(
                     compose(left,argument),
                     compose(tail,compose(right,argument))))))),
      constant argument))
```

As with the other higher order functions in this appendix, the only feasible ways to verify it would be either by formal proof or by some form of symbolic interpretation.

B.3 Replace

This code is needed in the discussion of assignment in Section 2.7.10 [Assignment], page 45. where it serves as a solution to equation *E0*. The idea is that the function takes an argument of the form `((`*locations*`,`*values*`),`*store*`)` and returns the store with the values stored at the locations indicated.

```
locations = compose(left,compose(left,argument))
values    = compose(right,compose(left,argument))
store     = compose(right,argument)

replace =

refer conditional(
   store,
   (
      conditional(
         compose(left,locations),
         (
            conditional(
               compose(right,locations),
               (
                  (hired meta)(
                     self,
                     couple(
                        (hired fan right)(locations,values),
                        (hired meta)(
                           self,
                           couple(
                              (hired fan left)(locations,values),
                              store)))),
                  couple(
                     (hired meta)(
                        self,
                        couple(
                           couple(compose(left,locations),values),
                           compose(left,store))),
                     compose(right,store)))),
            conditional(
               compose(right,locations),
               (
                  couple(
                     compose(left,store),
                     (hired meta)(
                        self,
                        couple(
                           couple(compose(right,locations),values),
                           compose(right,store)))),
                  values)))),
         (hired meta)(
            self,
            couple(couple(locations,values),constant (nil,nil)))))
```

B.4 Transition

This code is relevant to the discussion of `transfer` in Section 2.7.13.5 [Transfer], page 50, where its specification is described in detail. When this code is evaluated on a virtual code application f, the result is the code for a transition function that takes one configuration to the next in the course of evaluating a transfer function, as specified in equations *E7* to *E9*.

```
output_buffer = compose(left,argument)
input_buffer = compose(right,compose(right,argument))
active = compose(left,compose(right,argument))
state = compose(left,active)
output = compose(right,active)

transition =

bu(compose,refer) (hired bu(conditional,active))(
   (hired conditional)(
      constant input_buffer,
      bu(compose,(fan bu(hired meta,self))) (hired apply)(
         constant fan bu(couple,couple(output,output_buffer)),
         couple (fan bu(compose,couple))(
            couple(
               (hired apply)(
                  hired,
                  constant (state,compose(head,input_buffer))),
               constant compose(tail,input_buffer)),
            couple(
               (hired apply)(hired,constant(state,constant nil)),
               constant constant nil)))),
   constant compose(flat,compose(reverse,output_buffer)))
```

Appendix C Changes

This section is reserved for brief updates due to changes in the software that may be important enough to note temporarily until more thorough revisions to the document can be made.

The lack of content here indicates that the current version is either completely up to date or in such a sorry state of neglect that even this section is obsolete.

Appendix D External Libraries

Various functions are callable from virtual code applications by way of the `library` combinator as explained in Section 2.7.16.1 [Library combinator], page 59. An expression (shown in `silly` syntax) of the form `library('foo','bar') x` applies a function named `'bar'` from a library named `'foo'` to an argument `x`.

A brief overview of the libraries and functions can always be had by executing

```
$ avram --external-libraries
```

The listing displayed by this command may show some that are not included here if this version of the documentation is not current or your installation has been locally enhanced. It may also lack some that are documented here if your installation is not fully equipped.

Although the overview from the command line is adequate for a reminder, it is not informative enough to explain how each function should be used. The purpose of this section is to provide this information in greater detail.

Some general comments are applicable to all libraries.

Each library documented in this section can generate error messages in the event of exceptional conditions, that are documented individually. In addition to those, it's also possible for any library function to return error messages of

```
<'unrecognized library'>
<'unrecognized xxxx function name'>
```

where *xxxx* is the name of a library. These indicate either that the library name is invalid, or the library name is valid but the function name is invalid, or that they're both valid but the library wasn't detected on the host when `avram` was compiled. A virtual code application can always avoid these errors by testing for the availability of a function using the `have` combinator (Section 2.7.16.2 [Have combinator], page 60).

In addition, any library function that operates on numerical values or lists thereof can return these messages in cases of invalid input.

```
<'missing value'>
<'invalid value'>
<'bad vector specification'>
<'bad matrix specification'>
```

These messages indicate that an input parameter that was required to be a valid representation of a floating point number, a vector, or a matrix was something other than that (Section 3.1.4 [Type Conversions], page 72). The last could also occur if a parameter that is required to be a square matrix has unequal numbers of rows and columns.

D.1 bes

An interface to the Bessel functions as defined in the GNU Scientific Library (gsl) is available to virtual code applications by invoking a function of the form

```
library('bes',f)
```

where f is a character string identifying the Bessel function family. All functions in this library return a floating point number encoded as in Section D.11 [math], page 150.

D.1.1 Bessel function calling conventions

The virtual code interface simplifies the gsl C language API by excluding the facilities for error estimates, omitting certain array valued functions, and subsuming sets of related functions within common ones where possible.

The functions with names in the following group take an argument of the form `(n,x)`, where `n` identifies the member of the function family, and `x` is the argument to the function.

- `J` regular cylindrical Bessel functions
- `Y` irregular cylindrical Bessel functions
- `I` regular modified cylindrical Bessel functions
- `K` irregular modified cylindrical Bessel functions

For these functions, `n` can be either a natural number encoded as in Section 2.4 [Representation of Numeric and Textual Data], page 23, or a floating point number encoded as in Section D.11 [math], page 150. The latter case specifies functions of a fractional order. The relevant gsl function is called based on the value and type of the parameter.

Two further related families of functions follow the same calling convention.

- `Isc` scaled regular modified cylindrical Bessel functions
- `Ksc` scaled irregular modified cylindrical Bessel functions

The foregoing functions are related to those above by an exponential scale factor as documented in the gsl reference manual.

Functions with names in the following group also take an argument of the form `(n,x)`, but are not defined for fractional orders and so require a natural number for `n`.

- `j` regular spherical Bessel functions
- `y` irregular spherical Bessel functions
- `isc` regular modified spherical Bessel functions
- `ksc` irregular modified spherical Bessel functions

The functions in the remaining group follow idiosyncratic calling conventions.

- `zJ0`, `zJ1` These take a natural number `n` and return the `n`th root of the regular cylindrical Bessel functions of order 0 or 1, respectively.
- `zJnu` This takes a pair `(nu,n)` where `nu` is the (fractional) order of a regular cylindrical Bessel function, `n` is a natural number. It returns the `n`th zero of the function.
- `lnKnu` This takes a pair of floating point numbers `(nu,x)` where `nu` is the (fractional) order of an irregular modified cylindrical Bessel and `x` is the argument to the function, and it returns the natural log of the function.

D.1.2 Bessel function errors

Memory overflows and unrecognized function names can happen as with other library interfaces. A message of

```
<'bad bessel function call'>
```

means that invalid input parameters were given, such as a fractional order to a function family that is defined only for natural orders.

D.2 `complex`

Complex numbers are represented according to the ISO C standard as arrays of two IEEE double precision floating point numbers of 8 bytes each, with the number representing the real part first.

A small selection of operations on complex numbers is available by function calls of the form `library('complex',f)`. These functions are implemented by the host system's C library.

One example is `library('complex','create')` which takes a pair of floating point numbers (x,y) to a complex number whose real part is x and whose imaginary part is y. See Section D.11 [math], page 150 for information about constructing floating point numbers.

Other than that, the `complex` library functions `f` fall into three main groups, which are the real valued unary operations, the complex valued unary operations, and the complex valued binary operations. All of these operations are designated by their standard C names as documented elsewhere, such as the GNU `libc` reference manual, except as noted.

- real valued unary operations

```
creal  cimag  cabs  carg
```

- complex valued unary operations

```
ccos   cexp   clog   conj   csin   csqrt
ctan   csinh  ccosh  ctanh  casinh cacosh
catanh casin  cacos  catan
```

- complex valued binary operations

```
cpow  vid  bus  mul  add  sub  div
```

The last four correspond to the C language operators `*`, `+`, `-`, and `/` for complex numbers. The functions named `vid` and `bus` are similar to `div` and `sub`, respectively, but with the operands interchanged. That is,

```
library('complex','vid') (x,y)
```

is equivalent to

```
library('complex','div') (y,x)
```

All functions in this library taking complex numbers as input may also operate on real numbers, and binary operators can have either or both operands real. For real operands, a value of zero is inferred as the imaginary part. The result type of the function is the same regardless.

D.3 `fftw`

Some functions in the `fftw` fast Fourier transform library are callable by virtual code programs of the form `library('fftw',f)`, where `f` can be one of the following character strings.

`u_fw_dft` (uni-dimensional forward Discrete Fourier transform) `u_bw_dft` (uni-dimensional backward Discrete Fourier transform)

`b_fw_dft` (bi-dimensional forward Discrete Fourier transform)

`b_bw_dft` (bi-dimensional backward Discrete Fourier transform)

`u_dht` (uni-dimensional Discrete Hartley transform)

`b_dht` (bi-dimensional Discrete Hartley transform)

These stand for the discrete Fourier transform, in one dimension and two dimensions, either backward or forward, and the discrete Hartley transform in one dimension and two dimensions. The `fftw` library documentation (`http://www.fftw.org`) can give more information about the meaning of these transformations.

The interface is somewhat simplified compared to the API for the `fftw` C library because there are no considerations of memory management or planning, nor any provision for dimensions higher than two.

Furthermore, from the virtual side of the interface, these functions operate on lists rather than arrays. The one dimensional Fourier transforms take a list of complex numbers to a list of complex numbers (see Section D.2 [complex], page 137), and the one dimensional Hartley transforms take a list of reals to a list of reals (see Section D.11 [math], page 150). The two dimensional transforms are analogous but they take a matrix represented as a list of lists. Error messages pertaining to invalid input documented at the beginning of this section (Appendix D [External Libraries], page 135) are relevant.

Finally, unlike the native API for `fftw`, these transformations are scaled so that the backward transformation is the inverse of the forward, and the Hartley transformations are their own inverses (subject to roundoff error).

D.4 `glpk`

The `glpk` library (`ftp://ftp.gnu.org/pub/gnu/glpk/`) solves linear programming problems by the either the simplex algorithm or an interior point method.

The API for C client programs involves a complicated protocol with many optional settings, which is simplified for the virtual machine interface. Specifically, the library gives a choice of only two functions, which can be expressed in the following forms.

```
library('glpk','simplex')

library('glpk','interior')
```

These functions have the same calling convention and should return generally the same output for identical inputs, but differences in performance, precision, and maybe correctness can be expected. The remainder of this section applies to both of them.

D.4.1 `glpk` input parameters

The argument must be a triple of the form, `(`c`,(`m`,`y`))`, subject to the following specification.

- c is a list of cost function coefficients as floating point numbers (see Section D.11 [math], page 150). There should be one item of c for each variable in the linear programming problem (Note that there is no additive constant, which would require one extra).

The interpretation of *c* is that an assignment of non-negative values to the variables *x* is sought to make the vector inner product *c x* as small as possible.

- *m* is a sparse matrix represented as a list of triples in the form

  `<((i,j),a)...>`

  where *i* and *j* are row and column indices as natural numbers starting from 0 and *a* is a non-zero floating point number. The presence of a triple `((i,j),a)` in the list indicates that the *i*,*j*-th entry in the matrix has a value of *a*. Missing combinations of *i* and *j* indicate that the corresponding entry is zero.

  The interpretation of *m* is that together with *y* it specifies a system of equations the variables in the solution *x* must satisfy simultaneously, as explained below.

- *y* is a list of floating point numbers, with one number for each distinct value of *i* in *m*, above, needed to complete the equations.

  The interpretation of *y* is that in matrix notation, the condition $m\ x = y$ must be met by any acceptable solution *x*.

  To put it another way, for each distinct value of *i*, the *i*-th item of *y* has to equal the sum over all *j* of *xj a*, where *a* is the real number appearing in the triple `((i,j),a)` in *m*, if any, and *xj* is the *j*-th variable of the solution.

D.4.2 glpk output

If a solution meeting the constraints is found, it is returned as a list of pairs of the form `<(i,x)...>`, where each *i* is a natural number and each *x* is a floating point number giving the value obtained for the *i*-th variable numbered from zero. Any values of *i* that are omitted from the list indicate that the corresponding variable has a value of zero.

If no solution is found due to infeasibility or because `glpk` just didn't find one, an empty list is returned. The lack of a solution is not treated as an exceptional condition.

D.4.3 glpk errors

Possible error messages are

`<'bad glpk specification'>`

which means that the input did not conform to the description given above, and

`<'memory overflow'>`

It is not considered an exceptional condition for no feasible solution to exist, and in that case an empty list is returned.

The `glpk` documentation gives no assurance as to the correctness of reported solutions, so the user should also take the possibility of incorrect results into account.

D.4.4 Additional glpk notes

A sparse matrix representation of *m* is used because in practice most linear programming problems have very sparse systems of equations.

Only the constraint of non-negativity is admitted. Other constraints such as upper bounds must be effected through a change of variables if required.

The `glpk` library has a small memory leak, which `avram` corrects by methods described in Section 3.9.3.2 [Memory leaks], page 115.

D.5 gsldif

Numerical differentiation of a real valued function of a single real variable can be done by a library function of the form

```
library('gsldif',method)
```

where `method` is one of

- `'backward'`
- `'central'`
- `'forward'`
- `'t_backward'`
- `'t_central'`
- `'t_forward'`

D.5.1 gsldif input parameters

The argument to the functions with mnemonics of `backward`, `central` or `forward` is a pair `(`f`,`x`)`, where f is the virtual machine code for a real valued function of a real variable, and x is the input to f where the derivative is sought. Real numbers are represented according to Section D.11 [math], page 150.

The argument to the functions with mnemonics of `t_backward`, `t_central` or `t_forward` is a pair `((`f`,`t`),`x`)`, where f and x are as above, and t is a tolerance represented as a floating point number. The tolerance is passed through to the GNU Scientific library (GSL) differentiation routines.

When no tolerance is specified, the default is `1.0e-8`.

D.5.2 gsldif output

The result returned by `library('gsldif',method) (f,x)` or `library('gsldif',method) ((f,t),x)` is an approximation of the first derivative of f evaluated at x.

The result is obtained by the one of the GNU Scientific Library (GSL) functions for numerical differentiation that matches the virtual code function name. These functions are documented in the GSL reference manual. The three methods should have approximately the same results but may differ in numerical properties.

D.5.3 gsldif exceptions

An error message of

```
<'bad derivative specification'>
```

will be returned if the either the whole argument, f, or x is `nil`.

Any error message caused by the evaluation of f will propagate to the result.

D.5.4 Additional `gsldif` notes

The function *f* may be any expressible virtual machine code function that takes a real argument to a real result, including one that uses other library functions. However, if *f* passes functions to other library functions as arguments, there is a constant overhead in stack space for each level, and a remote possibility of a segmentation fault if they are very deeply nested.

Numerical instability is an issue for higher derivatives (i.e., differentiating a function that is obtained by differentiating another function). Some experimentation with larger tolerances may be needed.

D.6 `gslevu`

This library exports a pair of functions of the form

```
library('gslevu','accel')

library('gslevu','utrunc')
```

that take a list of real numbers *x* to a pair of real numbers `(`*s*`,`*e*`)`.

The idea is that *x* represents the first few terms of an infinite series whose sum converges, but only very slowly. The functions extrapolate an estimate of the infinite summation by the Levin u-transform as documented in the GNU Scientific Library reference manual.

For well behaved series, considerably fewer terms are needed for an accurate estimate than a direct summation would require.

D.6.1 `gslevu` calling conventions

The input to either of these functions is a list of real numbers represented as explained in Section D.11 [math], page 150.

The result is a pair `(`*s*`,`*e*`)` holding an estimate of the sum, *s*, and an estimate of the error in the sum, *e*, each being a real number.

Both functions compute the same sum, *s*, but the `utrunc` function is faster and more memory efficient, using a less trustworthy method of estimating the error.

D.6.2 `gslevu` exceptions

If an empty list is passed as a parameter to a function in this library, an error message of `<'empty gslevu sequence'>` is returned.

If there is insufficient memory, an error message of `<'memory overflow'>` is returned.

Other than that, no exceptional conditions are relevant other than the general ones documented at the beginning of Appendix D [External Libraries], page 135.

D.7 `gslint`

An interface to a selection of numerical integration routines from the GNU Scientific Library is provided by functions of the form

```
library('gslint',q)
```

where q can be one of `'qng'`, `'qng_tol'`, `'qagx'`, `'qagx_tol'`, `'qagp'`, or `'qagp_tol'`.

D.7.1 `gslint` input parameters

The library functions `qng` and `qagx` take an argument of the form `(`f`,(`a`,`b`))`, where f is a function to be integrated, a is the lower limit, and b is the upper limit, both limits being floating point numbers as in Section D.11 [math], page 150.

The `qng_tol` and `qagx_tol` functions take an argument of the form `((`f`,`t`),(`a`,`b`))`, where f, a, and b are as above, and t is a specified tolerance.

The `qagp` and `qagp_tol` functions take arguments of the form `(`f`,`p`)` and `((`f`,`t`),`p`)`, respectively, where f and t are as above, and p is an ordered list of real numbers specifying the limits of integration along with arbitrarily many intervening breakpoints.

The integrand f is expressed in virtual machine code, and takes a single real argument to a real result. The argument and result of f are required to be floating point numbers as described in Section D.11 [math], page 150. Any expressible function of this type is acceptable, even one defined in terms of other integrals, so that a double or triple integral can be expressed easily, albeit a costly computation. However, a constant overhead in stack space is required for each nested library function call, and there is currently no mechanism to prevent segmentation faults due to a stack overflow.

When no tolerance is specified, as with `qng`, `qagx`, and `qagp`, the tightest attainable tolerance is chosen by default, currently `2e-14`, in order find the most accurate result possible. A selection of progressively looser tolerances is tried automatically if the tightest one is not successful, stopping when either a solution is found or ten orders of magnitude are covered.

If a tolerance is explicitly specified, as with `qng_tol`, `qagx_tol` or `qagp_tol`, only that tolerance is tried.

D.7.2 `gslint` output

In all cases, if no exception occurs, the result returned is an approximation of the integral of f over the interval from a to b or from the first item of p to the last.

Results may differ in numerical properties depending on the integration method and the tolerance used.

- The `qagp`* and `qagx`* functions use an adaptive algorithm, whereas the `qng`* functions use a faster non-adaptive algorithm suitable only for smooth integrands.
- Faster and maybe more accurate results are obtained for discontinuous or non-differentiable integrands by the `qagp`* integration methods if the interior points in p are chosen to coincide with the discontinuities or corners.
- Larger tolerances are conducive to faster but less accurate results in most cases.

D.7.3 `gslint` exceptions

If an argument of an inappropriate form can be detected (such as an empty pair or one without floating point numbers), it causes an error message to be returned saying

```
<'bad integral specification'>
```

Error messages signalled by the integrand f may also be reported, as well as any message returned by `gsl_strerror`. A typical cause for a `gsl_strerror` message would be an explicitly specified tolerance that is too tight. An error message of

```
<'slow convergence'>
```

is returned in the event of excessively many function evaluations (currently 3600 at each tolerance level).

D.7.4 Additional `gslint` notes

The `qagx*` functions subsume the GSL variants `qags`, `qagiu`, `qagil`, and `qagi` for finite, semi-infinite, and infinite intervals, which are seleted as appropriate based on the limits of integration a and b.

The `qagp` function reverts to the `qagx` function if there are only two points given in p. Fewer than two will cause an error.

The library interface code relies on the standard `setjmp` utility found in the system header file `setjmp.h` to break out of integrals that don't converge after excessively many function evaluations. Non-termination has been an issue in the past with GSL integration routines for very badly behaved integrands, and the API provides no documented means for the user supplied integrand function to request a halt.

Although it is meant to be standard, a host without `setjmp` will cause `avram` to be configured to abort the application with an error message in the event of non-convergence. This behavior is considered preferable to the alternative of non-termination. Usually an effective workaround in such cases is to specify a sufficiently loose tolerance explicitly by using one of the `*_tol` library functions.

D.8 `harminv`

The `harminv` library decomposes a complex valued function of a discrete variable into a sum of decaying sinusoids given a finite sample. It uses a method with better accuracy and convergence than Fourier analysis or least squares curve fitting. More information is available at `http://ab-initio.mit.edu/wiki/index.php/Harminv`.

D.8.1 `harminv` input parameters

The virtual machine interface to the `harminv` library provides only a single function, callable as

```
library('harminv','hsolve')
```

The input to this function is an operand of the form

```
(signal,(fmin,fmax),nf)
```

where

- `signal` is a list of complex numbers containing samples of the function to be decomposed at equal time steps (Section D.2 [complex], page 137 and Section 2.4 [Representation of Numeric and Textual Data], page 23).
- `fmin` and `fmax` are the band limits expressed in units of inverse time steps as floating point numbers (Section D.11 [math], page 150).
- `nf` is the number of spectral basis functions expressed as a natural (Section 2.4 [Representation of Numeric and Textual Data], page 23).

If a value of 0 is specified for `nf` a default value of

```
min(300, (fmax - fmin) * n * 1.1)
```

is used, where `n` is the length of `signal`. The computation time increases cubically with `nf`.

D.8.2 harminv output

The result returned by a call to

```
library('harminv','hsolve')
```

with valid input (Section D.8.1 [harminv input parameters], page 143) is a list of similar tuples of the form

```
<(amplitude,frequency,decay,quality,error)...>
```

with all members being real valued except for the amplitudes, which are complex. Each tuple describes a function of the form

```
f(t) = A * sin (frequency * t + P) * exp (-decay * t)
```

such that the summation of these functions approximates the original given signal (Section D.8.1 [harminv input parameters], page 143). The real amplitude `A` and phase `P` are given by the modulus and argument of the complex amplitude returned in the result,

```
A = library('complex','cabs') amplitude
P = library('complex','carg') amplitude
```

in terms of the complex library functions (Section D.2 [complex], page 137). The error values are measures of the goodness of fit, and the quality factors are defined as

```
quality = (pi * |frequency| / decay)
```

It may be useful in some applications to ignore components with quality factors outside of a certain range.

D.8.3 harminv exceptions

Various exceptional conditions are possible with the `harminv` library interface, and one of the following messages could be returned. Each of them has the form of a list containing a single character string.

- `unrecognized harminv function name` is reported in case of a function call of the form `library('harminv',f)` where `f` is anything other than the character string `'hsolve'`, this being the only function in the library.
- `bad harminv function call` is reported if the input parameters don't meet the specifications described in Section D.8.1 [harminv input parameters], page 143, or if `fmin` is greater than `fmax`.
- `bad vector specification` could be the result of a list of real numbers rather than complex numbers being passed as a `signal`. Real numbers can be converted to complex numbers using the `create` function from the `complex` library (Section D.2 [complex], page 137).
- `memory overflow` can occur if `avram` is operating very close to the limit of host memory, or perhaps if infeasibly large values are passed as `nf`
- `counter overflow` is similar to a memory overflow

D.8.4 Additional `harminv` notes

The `harminv` library interface requires the `harminv` and `lapack` libraries to be installed on the host system, and also requires standard complex number support from the system's C library.

The author's installation of `avram` has been compiled against the Debian `harminv` development library package, which at this writing is unmaintained and is missing the necessary header file '`harminv-int.h`', without which compilation of files including '`harminv.h`' fails. Some headers from '`harminv.h`' have been copied directly into '`avram-x.x.x/src/harminv.c`' under the `avram` source tree to avoid this dependence, so that `avram` will compile correctly on a Debian system. These may need to be updated if necessary to track the `harminv` source.

D.9 `kinsol`

The `kinsol` library (`http://www.llnl.gov/CASC/sundials/`) contains sophisticated routines for non-linear optimization and constrained non-linear optimization, some of which are available to virtual code applications by way of functions expressed as shown.

```
library('kinsol',k)
```

The function name `k` is a string of the form `'`*xy*`_`*zzzzz*`'`. The field *zzzzz* specifies the optimization algorithm, which can be one of `dense`, `gmres`, `bicgs`, or `tfqmr`, following the names used by the API for `kinsol` in C. The field *y* determines the way gradients are obtained, which is either `j` for a user supplied Jacobian, or `d` for finite differences computed by `kinsol`. The remaining field *x* is either `c` for constrained optimization, or `u` for unconstrained. Hence, the whole function name can be one of sixteen possible alternatives.

```
cd_dense   cd_gmres   cd_bicgs   cd_tfqmr
ud_dense   ud_gmres   ud_bicgs   ud_tfqmr
cj_dense   cj_gmres   cj_bicgs   cj_tfqmr
uj_dense   uj_gmres   uj_bicgs   uj_tfqmr
```

More specific information about the optimization algorithms can be found in the `kinsol` documentation at the above address. Different algorithms may perform better on different problems.

D.9.1 `kinsol` input parameters

Functions whose names are of the form `xd_`*zzzzz* take an argument of the form `(`*f*`,(`*i*`,`*o*`))`, and functions whose names are of the form `xj_`*zzzzz* take an argument of the form `((`*f*`,`*j*`),(`*i*`,`*o*`))`. The parameters have these interpretations.

- *f* is a function to be optimized, expressed in virtual machine code. It takes a list of real numbers as input and returns a list of real numbers as output. The numbers must be in floating point format as described in Section D.11 [math], page 150.
- *j* is a function in virtual machine code that computes the Jacobian or partial derivatives of *f* for a given list of input numbers. The exact calling convention for *j* depends on the optimization algorithm selected, as explained below.
- *i* is a list of real numbers suitable as an input for *f*. The exact values of the numbers in *i* are not crucial but the length of *i* is taken as an indication of the required length

for any input list to *f*. In the case of constrained optimization problems (i.e., functions with names beginning with `c`), *i* must consist entirely of non-negative numbers.

- *o* is a list numbers indicating the "optimal" output from *f* in the sense described below (Section D.9.2 [kinsol output], page 146). Its length is taken to indicate the usual length of an output returned by *f*.

If the optimization problem is being solved by either the `cj_dense` or the `uj_dense` method, the Jacobian parameter *j* is expected to take a list *v* of real numbers the length of *i* as input and return a list of lists of reals as output. The numbers are represented as described in Section D.11 [math], page 150. The outer list in the output from *j* is required to be the length of *o*, while each inner list is required to be the length of *i*.

The output from *j* is interpreted as a matrix of the form described in Section 3.1.4.3 [Two dimensional arrays], page 74. The entry in row *m* and column *n* is the partial derivative (evaluated at *v*) of the *m*-th component of the output of *f* with respect to the *n*-th item of the input list.

For optimization problems being solved by the methods of `xj_gmres`, `xj_bicgs`, or `xj_tfqmr`, (i.e., where *x* is either `c` or `u`) the Jacobian function *j* follows a different convention that is meant to be more memory efficient. Given an argument of the form `(m,v)`, it returns only the *m*-th row of the matrix described above instead of the whole thing. The parameter *m* is a natural number less than the length of *o*, and *v* is a list of real numbers the length of *i* the same as above. The number *m* is encoded as described in Section 2.4 [Representation of Numeric and Textual Data], page 23.

D.9.2 `kinsol` output

The `kinsol` functions attempt to search the domain of *f* for a vector *v* the length of *i* to satisfy `f(v) = o` as closely as possible. In the case of constrained optimization, (i.e., functions whose names begin with `c`), only non-negative numbers are acceptable in *v*. The search for *v* will start in the vicinity of *i*. The value of *i* will therefore determine a unique solution if multiple solutions exist, and will save time if it is near a solution.

In some cases when a solution can't be found due to non-convergence, an empty list is returned. Non-convergence is not considered an exceptional condition. In all other cases where no exception occurs, the output from a `kinsol` function will be the list *v* of real numbers satisfying `f(v) = o` to the best possible tolerance.

D.9.3 `kinsol` exceptions

- Any error messages that may be generated in the course of evaluating the functions *f* and *j* will propagate to the result returned by the `kinsol` library functions.
- If there is insufficient memory to complete any operation, the result is a message of

  ```
  <'memory overflow'>
  ```

- If the argument to the library function (i.e., `(f,(i,o))` or `((f,j),(i,o))`) fails to meet the required specifications in a detectable way, the result will be a message of

  ```
  <'bad kinsol specification'>
  ```

- Any status returned by any `kinsol` API functions other than success or one of several types of non-convergence results in a message of

  ```
  <'kinsol error'>
  ```

D.9.4 Additional `kinsol` notes

When a user supplied Jacobian function j is specified, the solution is likely to be found faster and more accurately. The Jacobian should be given if an analytical form for f is known, from which the Jacobian can be obtained easily by partial differentiation. If the Jacobian is unavailable, a finite difference method implemented internally by `kinsol` is used as a substitute and will usually yield acceptable results.

Tolerances are not explicitly specified on the virtual side of the interface although the native `kinsol` API requires them. A range of tolerances over ten orders of magnitude is automatically tried before giving up.

Similarly to the `glpk` and `lpsolve` library interfaces (Section D.4 [glpk], page 138 and Section D.15 [lpsolve], page 160), the only expressible constraint through the virtual code interface is that all variables are non-negative. Arbitrary upper and lower bounds can be simulated by appropriate variable substitutions in the formulation of the problem.

The `kinsol` library natively requires a system function f with equally many inputs as outputs, and will search only for the input associated with an output vector of all zeros, but the virtual code interface relaxes these requirements by allowing a function that transforms between lists of unequal lengths, and will search for the input of f causing it to match any given "optimal" output o. These effects are achieved by padding the shorter of the two vectors transparently and subtracting the specified optimum from the result.

The `kinsol` library can be configured to use single precision, double precision, or extended precision arithmetic, but only a double precision configuration is compatible with `avram`. This condition is checked when `avram` is configured and it will not interface with alternative `kinsol` configurations.

The `kinsol` library has some more advanced features to which this interface doesn't do justice, such as preconditioning, scaling, solution of systems with band limited Jacobians, and concurrent computation.

D.10 `lapack`

An arsenal of weapons grade linear algebra functions from the `LAPACK` Fortran library is accessible to virtual code applications through library calls of the form

```
library('lapack',f)
```

Each library function `f` invokes a `LAPACK` function of the same name, but the calling conventions on the virtual side are an artifact of the interface requiring their own documentation.

Some functions that are part of `LAPACK` are not described here (mostly the so called computational and auxiliary routines, and anything in single precision), because they are not accessible by the virtual code interface.

D.10.1 `lapack` calling conventions

A table describing the inputs and outputs to the `lapack` library functions listed by their function names is given in this section. Some general points related to most of the functions are mentioned first.

- References to vectors, matrices, and packed matrices should be understood as their list representations explained in Section 3.1.4 [Type Conversions], page 72. Although `LAPACK` internally uses column order arrays, the virtual code library interface exhibits a matrix as a list of lists with one inner list for each row.
- Some functions require a symmetric matrix as an input parameter. Any input parameter that is required to be a symmetric matrix may be specified optionally either in square form or in triangular form as described in Section 3.1.4.3 [Two dimensional arrays], page 74. If a square matrix form is used, symmetry is not checked and the lower triangular portion is ignored.
- Some function names are listed in pairs differing only in the first letter. Function names beginning with `d` pertain to vectors or matrices of real numbers (Section D.11 [math], page 150), and function names beginning with `z` pertain to complex numbers (Section D.2 [complex], page 137). The specifications of similarly named functions are otherwise identical.

`dgesvx`

`zgesvx` These library functions take a pair `(`*a*`,`*b*`)` where *a* is an *n* by *n* matrix and *b* is a vector of length *n*. If *a* is non-singular, they return a vector *x* such that *a x* `=` *b*. Otherwise they return an empty list.

`dgelsd`

`zgelsd` These functions generalize those above by taking a pair `(`*a*`,`*b*`)` where *a* is an *m* by *n* matrix and *b* is a vector of length *m*, with *m* greater than *n*. They return a vector *x* of length *n* to minimize the magnitude of *b* `-` *a x*.

`dgesdd`

`zgesdd` These functions take a list of *m* time series (i.e., vectors) each of length *n* and return a list of basis vectors each of length *n*. The basis vectors span the set of time series in the given list according to the singular value decomposition (i.e., with the basis vectors forming a series in order of decreasing significance). The number of basis vectors is at most *min*`(`*m*`,`*n*`)` but could be less if the input time series aren't linearly independent. An empty list could be returned due to lack of convergence.

`dgeevx`

`zgeevx` These functions take a non-symmetric square matrix and return a pair `(`*e*`,`*v*`)` where *e* is a list of eigenvectors and *v* is a list of eigenvalues, both of which will contain only complex numbers. (N.B., both functions return complex results even though `dgeevx` takes real input.) They could also return `nil` due to a lack of convergence.

`dpptrf`

`zpptrf` These functions take a symmetric square matrix and return one of the Cholesky factors. The Cholesky factors are a pair of triangular matrices, each equal to the transpose of the other, whose product is the original matrix.

- If the input matrix is specified in lower triangular form, the lower triangular Cholesky factor is returned.

- If the input matrix is specified in square or upper triangular form, the upper triangular Cholesky factor is returned.
- In either case, the result is returned in triangular form.

`dggglm`

`zggglm` The input is a pair of matrices and a vector `(`(A`,`B`),`d`)`. The output is a pair of vectors `(`x`,`y`)` satisfying Ax `+` By `=` d for which the magnitude of y is minimal. The dimensions all have to be consistent, which means the number of rows in A and B is the length of d, the number of columns in A is the length of x, and the number of columns in B is the length of y.

`dgglse`

`zgglse` The input is of the form `((`A`,`c`),(`B`,`d`))` where A and B are matrices and c and d are vectors. The output is a vector x to minimize the magnitude of Ax `-` c subject to the constraint that Bx `=` d. The dimensions have to be consistent, which means A has m rows, c has length m, B has p rows, d has length p, both A and B have n columns, and the output x has length n. It is also a requirement that p `<=` n `<=` m `+` p.

`dsyevr` This function takes a symmetric real matrix and returns a pair `(`e`,`v`)` where e is a list of eigenvectors and v is a list of eigenvalues. Both contain only real numbers. This function is fast and accurate but not as storage efficient as possible. If there is insufficient memory, it automatically invokes `dspev`.

`dspev` This function takes a symmetric real matrix and returns a pair `(`e`,`v`)` where e is a list of eigenvectors and v is a list of eigenvalues. Both contain only real numbers. It uses roughly half the memory of `dsyevr` but is not as fast or accurate.

`zheevr` This function takes a complex Hermitian matrix and returns a pair `(`e`,`v`)` where e is a list of eigenvectors and v is a list of eigenvalues. The eigenvectors are complex but the eigenvalues are real.

- A Hermitian matrix has Aij equal to the complex conjugate of Aji.
- Although not exactly symmetric, a Hermitian matrix may nevertheless be given in either upper or lower triangular form.
- This function is faster but less storage efficient than `zhpev`, and calls it automatically if it runs out of memory.

`zhpev` This function has the same inputs and approximate outputs as `zheevr` but is slower and more memory efficient because it uses only packed matrices.

D.10.2 `lapack` exceptions

- Any of these functions can return a message of

  ```
  <'memory overflow'>
  ```

  if it runs out of memory.
- If the input parameters don't meet the specification, they can also return

  ```
  <'bad lapack specification'>
  ```

- Any unexpected behavior from the LAPACK Fortran functions or irregular status returned by them is reported by the message

  ```
  <'lapack error'>
  ```

  Getting to the bottom of it may require some debugging of the avram source code in the file 'lapack.c'.

D.10.3 Additional lapack notes

The functions dgesdd and zgesdd are an effective dimensionality reduction technique for a large database of time series. A set of basis vectors can be computed once for the database, and then any time series in the database can be expressed as a linear combination thereof. To the extent that the data embody any redundant information, an approximate reconstruction of an individual series from the database will require fewer coefficients (maybe far fewer) in terms of the basis than original length of the series.

The library functions dgelsd and zgelsd are good for finding least squares fits to empirical data. If the matrix parameter a is interpreted as a list of inputs and the vector parameter b as the list of corresponding output data from some unknown linear function of n variables f, then x is the list of coefficients whereby f achieves the optimum fit to the data in the least squares sense.

These functions solve a special case of the problem solved by dggglm and zggglm where the parameter B is the identity matrix. For the latter functions, the output vector y can be interpreted as a measure of the error, and B can be chosen to express unequal costs for errors at different points in the fitted function.

Cholesky decompositions obtained by dpptrf and zpptrf are useful for generating correlated random numbers. A population of vectors of uncorrelated standard normally distributed random numbers can be made to exhibit any correlations to order by multiplying all of the vectors by the lower Cholesky factor of the desired covariance matrix.

D.11 math

The math library exports functions that operate on IEEE double precision floating point numbers using the host system's C library. The numbers are represented natively as contiguous blocks of 8 bytes each, and on the virtual side as lists of eight character representations. (More explanation is given in Section 3.1.4 [Type Conversions], page 72.) These functions take the form

```
library('math',f)
```

where f is a character string identifying the function in most cases by its standard name in the C library.

D.11.1 math library operators

The unary operators take a single real number to a real result. They include

```
ceil  floor round trunc
sin   cos   tan   sinh   cosh   tanh
asin  acos  atan  asinh  acosh  atanh
exp   log   sqrt  cbrt   expm1  log1p  fabs
```

The binary operators take a pair of real numbers (x,y) to a single real number output. They include

pow hypot atan2 remainder bus vid add sub mul div

where the last four correspond to the C language operators +, -, *, and /. The functions named bus and vid are like the sub and div functions, respectively, with the order of the operands reversed, as explained in Section D.2 [complex], page 137.

The meanings of these operators are documented in the GNU libc reference manual or other C language references. They follow IEEE standards including proper handling of nan and infinity.

D.11.2 math library predicates

There is one binary predicate, islessequal, and several unary predicates, isinfinite, isnan, isnormal, isubnormal and iszero.

The predicate islessequal takes a pair of floating point numbers (x,y) as an argument, and returns nil for a false result and (nil,nil) for a true result.

The unary predicates have the obvious interpretations as classification functions, and should probably be used in preference to comparison with constants in case the representations aren't unique.

D.11.3 math library conversion functions

The conversion function strtod takes a string representing a floating point number in C format to its representation. This function is the primary means of creating or initializing floating point numbers in virtual code. A value of floating point 0.0 is returned if the string is not valid, but no exception is raised.

The conversion asprintf is similar to the one by that name in C, but requires 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 will be a character string expressing the number in the specified format.

D.11.4 math library exceptions

The most likely cause of an exception is an attempt to apply a math library function to nil or to an argument that doesn't represent a floating point number. In these cases, an error message of <'missing value'> or <'invalid value'> will be the result.

An error message of <'invalid asprintf() specifier'> is reported by the asprintf function if the format specifier pertains to a string, such as '%s'. This error is specifically trapped because the alternative would be a segmentation fault. Otherwise, invalid format specifiers are not detected or reported.

Error messages of <'invalid text format'> can be generated by conversion functions if any parameters that are meant to be character string representations are something else.

There is always a chance of a <'memory overflow'> error if there is insufficient memory to allocate a result.

D.11.5 Additional math library notes

Floating point exceptions such as division by zero are not specifically reported as exceptions, but invalid computations can be detected by the propagation of nan into the result, following standard conventions.

The C function feclearexcept (FE_ALL_EXCEPT) is called before every floating point operation so that no lingering exception flags can affect it.

There is no library predicate for exact comparison of floating point numbers, but none is required because the virtual machine's compare combinator will work on their representations as it will on any other data. The usual caveats apply with regard to comparing floating point numbers in the presence of roundoff error.

D.12 mtwist

The mtwist library interfaces to a random number generator based on the Mersenne Twistor algorithm. The algorithm has good properties but is not meant to be cryptographically secure. The library functions are of the form

```
library('mtwist',f)
```

where f is one of the followng character strings.

```
bern   u_cont   u_disc   u_path   u_enum   w_disc   w_enum
```

Formally they are not mathematical functions because their results depend on a pseudorandom number that is not uniquely determined by their arguments. The numbers are generated deterministically in a sequence starting from a seed derived from the system clock at the time avram is launched, and each call uses the next number in the sequence. In so doing, it simulates a random draw from a uniformly distributed population.

D.12.1 mtwist calling conventions

All of the functions in this library simulate a random draw from a distribution. There is a choice of distribution statistics depending on the function used.

bern
: takes a floating point number p between 0 and 1, encoded as in Section D.11 [math], page 150, and returns a boolean value, either (nil,nil) for true or nil for false. A true value is returned only if a random draw from a uniform distribution ranging from 0 to 1 is less than p. This function therefore simulates a draw from a Bernoulli distribution. A nil value of p is treated as 1/2.

u_cont
: takes a floating point number x as an argument, and returns a random draw from a continuous uniform distribution ranging from 0 to x. A nil value of x is treated as unity.

u_disc
: simulates a draw from a uniform discrete distribution whose domain is the set of natural numbers from 0 to n - 1. The number n is given as a parameter to this function, and the retuned value is the draw.
  - The returned value will have at most 64 bits regardless of n.
  - Natural numbers are encoded as described in Section 2.4 [Representation of Numeric and Textual Data], page 23.

- If a value of 0 is passed for n, the full 64 bit range is used.

`u_path` takes a pair of natural numbers `(`n`,`m`)` and returns a randomly chosen tree (Section 2.1 [Raw Material], page 19) with 1 leaf and n non-leaves each having either a left or a right descendent but not both. The number m constrains the result to fall within the first m - 1 trees of this form enumerated by exhausting all possibilities at lower levels before admitting a right descendent at a higher level. Within these criteria, all possible results are equally probable. Both numbers are masked to 64 bits, but if m is zero, it is treated as 2^n.

`u_enum` simulates a random draw from a uniform discrete distribution whose domain is enumerated. The argument to the function is a non-empty list, and the result is an item selected from the list, with all choices being equally probable.

`w_disc` simulates a random draw from a non-uniform, or “weighted” discrete distribution whose domain is a set of consecutive natural numbers starting from zero. The argument to the function is a list giving the probability of each outcome starting from zero as a floating point number. Probabilities must be non-negative but needn’t be normalized.

`w_enum` simulates a random draw from a non-uniform, or “weighted” discrete distribution with an arbitrary domain enumerated in the argument. The argument is a list of pairs `<(`x`,`p`)..>`, where x is a possible outcome and p is its probability. The result returned is one of the values of x from the input list chosen at random according to the associated probability. Probabilities must be non-negative but needn’t be normalized.

D.12.2 `mtwist` exceptions

- `<'memory overflow'>` can be returned if there is insufficient memory to allocate a result.
- Messages of `<'missing value'>` and `<'invalid value'>` can be returned if any floating point argument is `nil` or is not a valid floating point number, unless there is a designated default interpretation for `nil` as in `bern` and `u_cont`.
- A message of `<'bad mtwist specification'>` is returned if an argument to the `bern` function is not in the range of 0 to 1, or if any probability passed to the `w_*` functions is negative.
- A message of `<'draw from empty list'>` is returned if an argument to the `*_enum` functions is `nil` or if an argument to `w_enum` contains `nil`.

D.12.3 Additional `mtwist` notes

Although the `mtwist` library is “external”, it requires no special configuration on the host because the uniform variate generator in the form developed by its original authors is short and elegant enough to be packaged easily within the `avram` distribution. All further embellishments are home grown despite the advice at the end of Section 3.9.2 [Implementing new library functions], page 112.

The `u_path` function is intended to allow sampling from a large population in logarithmic time when it is stored in a balanced tree. A left-heavy tree should be constructed initially with the data items all at the same level. Thereafter, a result returned by `u_path` with the appropriate dimensions can be used as an index into the tree for fast retrieval by the virtual machine's `field` combinator (Section 2.7.8.1 [Field], page 42).

The last three functions, `u_enum`, `w_disc`, and `w_enum` use an inversion method with a binary search. The first draw from a given list will take a time asymptotically proportional to the length of the list, but subsequent draws from the same list are considerably faster due to a persistent cache maintained transparently by `avram`. For lists whose length is up to 2^16, the time required for a subsequent draw consists mainly of constant overhead with a small logarithmic component in the length of the list. For longer lists, the time ramps up linearly by a small factor.

Information allowing fast draws from up to sixteen lists can be cached simultaneously. If an application uses more than sixteen, the cached data are replaced in first-in first-out order. The size of the cache and the maximum list length for logarithmic time access can be adjusted easily by redefining constants in '`mtwist.c`' under the `avram` source tree, but will require recompilation.

D.13 `minpack`

The `minpack` library contains functions to solve non-linear optimization and least squares problems. The functions can be expressed as

```
library('minpack',f)
```

where `f` can be one of `'hybrd'`, `'hybrj'`, `'lmder'`, `'lmdif'`, or `'lmstr'`, following the names of the underlying Fortran subroutines.

D.13.1 `minpack` calling conventions

The `minpack` library solves a similar problem to that of the `kinsol` library (Section D.9 [kinsol], page 145), and the two libraries have identical calling conventions at the level of the virtual code interface.

The `hybrd` and `lmdif` functions take input arguments of the form `(`*f*`,(`*i*`,`*o*`))`, whereas `hybrj`, `lmder`, and `lmstr` take arguments of the form `((`*f*`,`*j*`),(`*i*`,`*o*`))`. The interpretations of these parameters are explained in Section D.9.1 [kinsol input parameters], page 145.

For the `lmstr` function, the Jacobian function *j* takes an argument `(`*m*`,`*v*`)` and returns only the *m*-th row of the Jacobian matrix. For `lmder` and `hybrj`, the Jacobian function takes only an input vector *v* and returns the whole matrix. These specifications are also explained further in relation to the `kinsol` library.

The output from any minpack function is a vector *v* satisfying *f*`(`*v*`) =` *o* to the best possible tolerance if a solution is found. A range of tolerances over ten orders of magnitude is sampled starting from `1e-15`. If no solution is found, an empty list is returned.

D.13.2 `minpack` exceptions

- A message of `<'memory overflow'>` is possible any time `minpack` runs out of memory.
- A message of `<'bad minpack specification'>` will be returned if an input argument recognizably violates the required specification.

- The <'minpack error'> message is returned in the event of any unexpected behavior or irregular status from the API.
- Any error messages reported by the system function *f* or the Jacobian function *j* are propagated to the result.

D.13.3 Additional `minpack` notes

The `lm*` functions are better suited to problems in which the system function *f* has more outputs than inputs, and the `hybr*` functions are better suited to the alternative. If either is called when the other is more appropriate, the job is handed off to the other automatically.

The `lmstr` function is more memory efficient than the others because it doesn't compute the whole Jacobian matrix at once. Any of the `lm*` functions is more memory efficient than the `kinsol` equivalent when the output list is sufficiently longer than the input list.

Unlike `kinsol`, there is no provision in `minpack` for constrained optimization.

The `minpack` documentation doesn't state whether it's re-entrant, but the odds are against it unless it uses no storage outside the user provided work areas. If it isn't re-entrant, anomalous effects could occur when a virtual code function being optimized calls another `minpack` function. A workaround would be to use an equivalent `kinsol` function, which is re-entrant by design.

The `avram` configuration script searches for a C header file '`minpack.h`' on the host system in order to build an interface to this library. This file is specific to the Debian `minpack-dev` package and is not part of the upstream Fortran source. Configuring `avram` with an interface to the `minpack` library on a non-Debian system may require the administrator to retrieve the header file manually from the Debian archive and place it under '`/usr/include`' before running the configuration script (in addition to installing the `minpack` library itself, of course).

D.14 `mpfr`

The `mpfr` library provides a rich assortment of floating point operations on arbitrary precision numbers (`http://www.mpfr.org`). These numbers are represented in a format that is not binary compatible with the standard IEEE floating point number format used by other libraries, but they offer superior numerical stability suitable for many ill conditioned problems.

The virtual code interface to the `mpfr` library follows the native API to the extent of using the same names for most operations, but excludes features pertaining to i/o, mutable storage, and memory management.

The `mpfr` library functions are invoked by an expression of the form

```
library('mpfr',f)
```

Aside from a few exceptions as noted, `f` is a character string derived from the name of the related function from the `mpfr` C library as documented at the above address, but without the `mpfr_` prefix.

The full complement of available functions is documented in the remainder of this section.

- References to natural numbers pertain to the list representation described in Section 2.4 [Representation of Numeric and Textual Data], page 23.

- All functions that perform rounding use a mode of `GMP_RNDN` for rounding to nearest, which is not explicitly specified on the virtual side.

D.14.1 `mpfr` binary operators

Functions with these names take a pair of `mpfr` numbers (x,y) and return an `mpfr` number as a result.

- `add`
- `sub`
- `mul`
- `div`
- `pow`
- `atan2`
- `hypot`
- `min`
- `max`
- `vid`
- `bus`

Their semantics are similar to those listed in the `mpfr` documentation, with some minor qualifications.

- Unlike the native API, there is no third argument to which the result is assigned, because the result is the returned value.
- The precision of the result is the greater of the two precisions of the input numbers x and y.
- The `vid` and `bus` functions are added features of the virtual code interface, corresponding to division and subtraction with the order of the operands reversed, as explained in Section D.2 [complex], page 137.

Mathematically it might make more sense for the precision of the result to be the lesser of the two input precisions, but this way is more convenient for virtual code programs that perform binary operations on their input with hard coded constants, because it makes one size fit all.

D.14.2 `mpfr` unary operators

Functions with these names take a single `mpfr` number as an argument and return a single `mpfr` number as a result.

```
sqr     sqrt    cbrt    neg     abs     log
log2    log10   exp     exp2    exp10   cos
sin     tan     acos    asin    atan    cosh
sinh    tanh    acosh   asinh   atanh   lngamma
expm1   eint    gamma   erf     log1p   nextbelow
ceil    floor   round   trunc   frac    nextabove
erfc
```

The semantics of these functions are similar to those of their counterparts in the native API, with these provisions.

- The precision of the result is the precision of the argument.
- There is no second argument for assigning the result.
- The `nextabove` and `nextbelow` functions do not modify their arguments in place, but return a freshly allocated result like all other functions.

D.14.3 `mpfr` binary operators with a natural operand

Functions with these names take an argument of the form `(`x`,`n`)`, where x is an `mpfr` number and n is a natural number.

- `root`
- `pow_ui`
- `mul_2ui`
- `div_2ui`
- `grow`
- `shrink`

The last two are specific to the virtual code interface, having no counterpart in the native API of the `mpfr` library. The `grow` function returns a copy of x with its precision increased by n bits, and the `shrink` function returns a copy of x with its precision reduced by n bits.

- The precisions are silently capped at the maximum or floored at the minimum allowable precisions if necessary.
- Increasing the precision by the `grow` function does not directly cause a more accurate result to be computed, but only pads an existing number with zeros.
- Decreasing the precision by the `shrink` function does not prevent valid bits from being discarded.

The appropriate way to use `grow` is to grow the precision of an operand before applying an operator to it, which will cause the result to be computed to the full precision. This capability is suitable for algorithms that iterate over increasing precisions until a stopping criterion is met.

D.14.4 `mpfr` binary predicates

These predicates take a pair of `mpfr` numbers `(`x`,`y`)` as arguments and perform a logical operation. If the result is true, they return `(nil,nil)`, and if it's false, they return `nil`.

- `equal_p`
- `unequal_abs`
- `greater_p`
- `greaterequal_p`
- `less_p`
- `lessequal_p`
- `lessgreater_p`

The name of the function `unequal_abs`, for comparing absolute values, has been changed from `mpfr_cmpabs` to avoid confusion with the virtual machine's `compare` combinator. The `compare` combinator returns a `(nil,nil)` result (i.e., true) if the operands are equal and a `nil` result if they're unequal, opposite from `unequal_abs`.

D.14.5 `mpfr` unary predicates

Each of these predicates takes an `mpfr` number as an argument and performs a logical operation. If the result is true, it returns `(nil,nil)`, and otherwise it returns `nil`.

- `nan_p`
- `inf_p`
- `number_p`
- `zero_p`
- `integer_p`

D.14.6 `mpfr` constants

Each of these functions takes a natural number as an argument specifying a precision, and returns a mathematical constant evaluated to that precision.

- `const_log2`
- `pi`
- `const_catalan`
- `inf`
- `ninf`
- `nan`

The name of the constant `pi` has been shortened from `mpfr_const_pi`. The functions `inf` and `ninf` return infinity and negative infinity, respectively.

The encoding of `nan`, used to represent the results of undefined computations such as division by zero, is not unique even for a fixed precision. Applications should test for undefined results using `nan_p` rather than by comparing a result to a hard coded `nan` (Section D.14.5 [mpfr unary predicates], page 158).

D.14.7 `mpfr` functions with miscellaneous calling conventions

Some functions listed below don't conform to any of the previously mentioned calling conventions.

`eq` This is a ternary operator taking a triple `(`*prec*`,(`x`,`y`))`, where *prec* is a natural number and x and y are `mpfr` numbers. It returns a result of `(nil,nil)` (i.e., true) if the numbers agree up to the specified precision measured in bits, and returns `nil` otherwise.[1]

[1] a potentially useful tool for algorithms concerned with numerical approximations despite its inexplicable malignment in the `mpfr` documentation

`urandomb` This function takes a natural number specifying a precision and returns a uniformly distributed pseudo-random number of that precision between 0 and 1.

`prec` This function takes an `mpfr` number and returns a natural number as a result, which is the precision of the argument in bits.

`sin_cos` This function takes an `mpfr` number *z* as an argument and returns a pair of `mpfr` numbers (*x*,*y*) as a result, where *x* is the sine of *z* and *y* is the cosine. The precisions of the results are the same as the precision of the argument.

D.14.8 `mpfr` conversion functions

The functions described in this section convert between `mpfr` numbers and character strings, naturals, or standard IEEE floating point format (in their list representations). Where these functions have similar or equivalent counterparts in the `mpfr` library's native API, the names have been changed for mnemonic reasons.

`dbl2mp` The input is a standard floating point number as in Section D.11 [math], page 150. The result is an `mpfr` number equal to the input with a fixed precision, currently set to 160 bits.

`mp2dbl` The input is an `mpfr` number, and the output is the best possible approximation to it by a standard a double precision number.

`str2mp` The input is a pair (*prec*,*s*), where *prec* is a natural number specifying the precision, and *s* is a string expressing a floating point number in C format. The output is an `mpfr` number with the specified precision.

`mp2str` The input is an `mpfr` number, and the output is a character string expressing the number in exponential decimal notation. Sufficiently many decimal digits are included in the string to express the full precision.

`nat2mp` The input is a natural number represented as described in Section 2.4 [Representation of Numeric and Textual Data], page 23, and the output is an `mpfr` number of sufficient precision to express the natural number exactly.

The `mp2str` function enhances the native `mpfr_get_str` function by properly formatting the output string rather than only listing the digits of the mantissa.

The `nat2mp` function does not rely on the `mpfr` native integer conversion functions, so natural numbers with any number of bits up to `MP_PREC_MAX` can be used losslessly. There is currently no conversion in the other direction.

D.14.9 `mpfr` exceptions

- A message of `<'memory overflow'>` is possible any time `mpfr` runs out of memory.
- A message of `<'bad mpfr specification'>` will be returned if an input argument recognizably violates the required specification.
- The `<'mpfr error'>` message is returned in the event of any unexpected behavior or irregular status from the API.
- The message of `<'mpfr overflow'>` can be cause by the `nat2mp` function if a natural number has too many bits to be represented exactly as an `mpfr` number.

D.14.10 Additional `mpfr` notes

The `eq` and `urandomb` functions depend not only on the `mpfr` library but on the `gmp` library (`http://ftp.gnu.org/gnu/gmp`). It's possible for them to be unavailable on a host without `gmp` even if the rest of the `mpfr` library is properly configured.

The file `mpfr.c` in the `avram` source tree exports a couple of functions that may be of use to C hackers interested in further development of `avram` with `mpfr`. The functions `avm_mpfr_of_list` and `avm_list_of_mpfr` convert between the native representation for `mpfr` numbers and the caching list representation used by `avram` (Section 3.1.4 [Type Conversions], page 72). This conversion is non-trivial because the numbers are not stored contiguously.

D.15 `lpsolve`

This library interface exports functions to solve linear programming and mixed integer programming problems using the `lpsolve` package documented at

`http://lpsolve.sourceforge.net/5.5/`.

Of the two linear programming solvers currently interfaced with `avram`, this one is believed to be the more robust.

D.15.1 `lpsolve` calling conventions

The library is able to solve linear and mixed integer programming problems, depending on which function is selected. The function to call the linear programming solver is of the form

- `library('lpsolve','stdform')`

and the mixed integer programming functions are of the form

- `library('lpsolve','iform')`
- `library('lpsolve','bform')`
- `library('lpsolve','biform')`

The argument to the `stdform` function represents a triple `(`c`,(`m`,`y`))`, which has the same interpretation described in Section D.4.1 [glpk input parameters], page 138. The arguments to the `iform`, `bform`, and `biform` functions are tuples `(`i`,(`c`,(`m`,`y`))) (`b`,(`c`,(`m`,`y`)))`, and `((`b`,`i`),(`c`,(`m`,`y`)))`, respectively, where c, m, and y are as above, and

- b is a list of binary variable column indices
- i is a list of integer variable column indices

where column indices pertain to the constraint matrix, and are numbered from zero. Specifying some or all variables as integers directs the solver to seek only solutions in which those variables have integer values, and specifying any as binary directs the solver to seek only solutions in which those variables have values of zero or one. The IEEE floating point representation is used for all variables regardless (Section D.11 [math], page 150).

D.15.2 `lpsolve` return values

If a feasible and optimal solution is found, a list of values for the variables is returned in the form `<(`*i*`,`*x*`)...>`, where i is a natural number and x is a floating point number giving the value of the i-th variable numbered from zero. Values of x equal to zero are omitted.

D.15.3 `lpsolve` errors

If any calling conventions are not followed, an exception is raised and a diagnostic message of `bad lpsolve problem specification` is reported. If no feasible solution can be found, no exception is raised but an empty list is returned.

D.16 `rmath`

A selection of mathematical and statistical functions from the GNU R math library has a virtual code interface of the form

```
library('rmath',f)
```

where `f` is a character string derived from the name of a function in the C language API described in the manual '`R-exts.pdf`', available at `http://www.r-project.org`.

Every function in the library returns a real result in the form of Section D.11 [math], page 150, but functions differ in the argument types. The arguments are tuples of real numbers and booleans that also closely follow the native API as explained below.

D.16.1 `rmath` statistical functions

Functions for evaluating random draws, density, cumulative probability and inverse cumulative probability are provided for some of the more frequently used probability distributions, which are chi-squared, non-central chi-squared, exponential, lognormal, normal, poisson, Student's t, and uniform.

Each distribution is known by an abbreviated name and specified by one or two real parameters as listed below. Names of distributions in this table form the stem of a library function name. The names of the parameters such as *mu* and *sigma* are not explicitly mentioned when invoking the functions, but are listed here for reference. The precise definitions of the distribution functions and interpretations of these parameters can be found in standard texts on probability and statistics.

`chisq`	*df*
`nchisq`	*df*, *lambda*
`exp`	*scale*
`lnorm`	*logmean*, *logsd*
`norm`	*mu*, *sigma*
`pois`	*lambda*
`t`	*n*
`unif`	*a*, *b*

The virtual code interface follows a naming convention similar to the native API, in that function names beginning with `r` represent random draws from a distribution, with the argument to the function being the parameters specifying the distribution. Functions in this first group return a random draw from a distribution described by a single real parameter.

- `rchisq`
- `rexp`
- `rpois`
- `rt`

These next functions return random draws from distributions specified by a pair of parameters, (x,y).

- `rnchisq`
- `rlnorm`
- `rnorm`
- `runif`

Functions whose names begin with `d` evaluate the probability density of a distribution at a given point. They require at least two real arguments, the first being the point whose probability density is sought, and the remaining ones being the parameters that specify the distribution. A boolean operand, which is `nil` for false and `(nil,nil)` for true, requests the logarithm of the density when true.

Functions with names in the following group take a triple with two real operands and a boolean, (x,(y,a)), and return a probabiity density.

- `dchisq`
- `dexp`
- `dpois`
- `dt`

The next functions pertain to distributions requiring two paramters to specify them, so they take a quadruple with three real operands and a boolean, (x,(y,(z,a))).

- `dnchisq`
- `dlnorm`
- `dnorm`
- `dunif`

Functions whose names begin with `p` or `q` obtain cumulative probabilities or inverse cumulative probabilities respectively for a specified distribution. They require one real operand to identify the point whose probability or inverse probability is sought, and other real operands to parameterize the distribution, as above. There are also two boolean operands. The first is true in order to request a probability or inverse probability with respect to the lower tail as opposed to the upper, and the other is true to indicate that probabilities are to be expressed logarithmically.

The argument to these functions is a quadruple with two real operands and two booleans, (x,(y,(a,b))).

- `pchisq`, `qchisq`
- `pexp`, `qexp`

- ppois, qpois
- pt, qt

The remaining functions pertain to distributions parameterized by two real operands. These take a quintuple with three real operands and two booleans, $(x,(y,(z,(a,b))))$.

- pnchisq, qnchisq
- plnorm, qlnorm
- pnorm, qnorm
- punif, qunif

D.16.2 rmath miscellaneous functions

Some less frequently used real valued mathematical functions are also accessible by the rmath library interface. The functions with names in this group take a single real operand.

gammafn	lgammafn	digamma
trigamma	tetragamma	pentagamma

The ones in this group take a pair of real operands (x,y).

beta	lbeta	bessel_j	bessel_y

Those remaining take a triple of real operands $(x,(y,z))$.

bessel_i	bessel_k

An alternative and better documented selection of Bessel functions is provided by the bes library interface (Section D.1 [bes], page 135).

D.16.3 rmath exceptions

The only exceptional condition specific to the rmath library interface is associated with the message <'bad rmath specification'>, which means that a tuple given as an argument has the wrong number or types of operands.

D.17 umf

Systems of equations described by sparse matrices (i.e., matrices containing mostly zeros) arise in certain practical problems. The usual array representation in which zeros are explicitly stored would be prohibitive for large matrices occurring in many problems of interest. A more sophisticated approach is used by the umf library to manage memory efficiently, which is documented at http://www.cise.ufl.edu/research/sparse/SuiteSparse/current/SuiteSparse/UMFPA

A virtual code interface to functions for solving sparse systems of equations by these methods is afforded by library functions of the form

```
library('umf',f)
```

where the library function name, f is a character string of the form *tt_m_rrr*.

- *tt* can be either di for real matrices, or zi for complex.
- *m* can be one of a, t, or c for solving a system given either by a matrix, its transpose, or its conjugate transpose, respectively, corresponding to mnemonics A, Aat and At used in the C language API.

- *rrr* is either `trp` or `col`, to indicate a sparse matrix expressed either as a list of triples, or in packed column form, as documented below.

The complete set of function names for this library interface is as follows.

```
di_a_trp    di_a_col    zi_a_trp    zi_a_col
di_t_trp    di_t_col    zi_t_trp    zi_t_col
                        zi_c_trp    zi_c_col
```

Not all combinations are represented, because the conjugate transpose is relevant only to complex matrices.

D.17.1 `umf` input parameters

For a square matrix A and a column vector b, the `umf` functions find the solution x to the matrix equation $M\ x = b$, where M is either A, the transpose of A, or its conjugate transpose. As noted above, the choice is determined by whether the the function name is of the form `*_a_*`, `*_t_*`, or `*_c_*` respectively.

The argument to any of these functions is a pair `(`A`,`b`)`, where A represents the matrix mentioned above and b represents the column vector.

The parameter b is required to be a list of numbers whose length matches the number of rows in the matrix. The numbers are either real numbers for the `di_*` functions (Section D.11 [math], page 150), or complex for the `zi_*` functions (Section D.2 [complex], page 137).

There is a choice of representations for the parameter A, depending on whether the function being called is one of the `*_trp` functions or one of the `*_col` functions.

For the `*_trp` functions, A is represented as a non-empty list of triples `<((`i`,`j`),`v`)...>`, where each item of the list corresponds to a non-zero entry in the matrix.

- The parameters i and j are natural numbers as in Section 2.4 [Representation of Numeric and Textual Data], page 23.
- The value v is a real number for the `di_*_trp` functions or a complex number for the `zi_*_trp` functions.
- The presence of a triple $((i,j),v)$ in the list signifies that the i,j-th entry in the matrix A (numbered from zero) has a value of v.

For the `*_col` functions, the representation of A is more complicated but has a slight advantage in memory usage. It may also have an advantage in speed unless more time is wasted on the virtual side transforming a matrix to this representation than it saves.

In this case, A is represented by a triple of the form `((`p`,`i`),`v`)`. The parameters p and i are lists of natural numbers. The parameter v is a list of real numbers for the `di_*_col` functions and complex numbers for the `zi_*_col` functions. They have the following interpretations.

- v is the list of non-zero entries in the matrix in column major order.
- i has the same length as v, and each item of i is the row index of the corresponding item in v, numbered from zero.
- p has the length of the number of columns in the matrix, and each item identifies the starting position of a column in v and i, numbered from zero.

The first item of p is always zero. Further explanation of this format in terms of an array representation can be found in the file 'UMFPACK_UserGuide.pdf', available from the umf library home page at http://www.cise.ufl.edu/research/sparse/SuiteSparse/current/SuiteSparse/.

D.17.2 umf output

If no exception occurs, the solution x to the matrix equation $M\ x = b$ noted previously will be returned if one exists.

The solution is represented as either a list of real numbers as in Section D.11 [math], page 150, or a list of complex numbers as in Section D.2 [complex], page 137. Real numbers are returned by the `di_*` functions, and complex numbers are returned by the `zi_*` functions.

If no solution exists due to a singular matrix, an empty list is returned. The lack of a solution isn't treated as an exceptional condition.

D.17.3 umf exceptions

If an exceptional condition arises from the use of this library, one of the following lists of character strings may be returned as the function result.

- `<'memory overflow'>` means the library function ran out of memory, most likely due to a matrix being too large.
- `<'bad umf specification'>` means an input parameter didn't conform to the appropriate format described above (Section D.17.1 [umf input parameters], page 164)
- `<'umf error'>` covers any unexpected behavior or abnormal status returned by any function from the C language API.

For the `*_trp` functions. A non-square matrix will cause the second exception above. For the `*_col` functions, a non-square matrix will cause the third exception or cause an empty result to be returned.

The exceptions noted at the beginning of this section (Appendix D [External Libraries], page 135) are also possible.

D.17.4 Additional umf notes

The C language API to umf provides very many less frequently used features that are not part of the virtual code interface, some of which could be added by minor modifications to the file 'umf.c' in the avram source tree.

A set of `dl_*` and `zl_*` functions orthogonal to those presently accessible would enable matrices having billions of rows or columns by using long integers, but memory requirements on the virtual code side for problems of that scale are probably prohibitive for the foreseeable future.

GNU GENERAL PUBLIC LICENSE

Version 2, June 1991

Preamble

The licenses for most software are designed to take away your freedom to share and change it. By contrast, the GNU General Public License is intended to guarantee your freedom to share and change free software—to make sure the software is free for all its users. This General Public License applies to most of the Free Software Foundation's software and to any other program whose authors commit to using it. (Some other Free Software Foundation software is covered by the GNU Library General Public License instead.) You can apply it to your programs, too.

When we speak of free software, we are referring to freedom, not price. Our General Public Licenses are designed to make sure that you have the freedom to distribute copies of free software (and charge for this service if you wish), that you receive source code or can get it if you want it, that you can change the software or use pieces of it in new free programs; and that you know you can do these things.

To protect your rights, we need to make restrictions that forbid anyone to deny you these rights or to ask you to surrender the rights. These restrictions translate to certain responsibilities for you if you distribute copies of the software, or if you modify it.

For example, if you distribute copies of such a program, whether gratis or for a fee, you must give the recipients all the rights that you have. You must make sure that they, too, receive or can get the source code. And you must show them these terms so they know their rights.

We protect your rights with two steps: (1) copyright the software, and (2) offer you this license which gives you legal permission to copy, distribute and/or modify the software.

Also, for each author's protection and ours, we want to make certain that everyone understands that there is no warranty for this free software. If the software is modified by someone else and passed on, we want its recipients to know that what they have is not the original, so that any problems introduced by others will not reflect on the original authors' reputations.

Finally, any free program is threatened constantly by software patents. We wish to avoid the danger that redistributors of a free program will individually obtain patent licenses, in effect making the program proprietary. To prevent this, we have made it clear that any patent must be licensed for everyone's free use or not licensed at all.

The precise terms and conditions for copying, distribution and modification follow.

TERMS AND CONDITIONS FOR COPYING, DISTRIBUTION AND MODIFICATION

1. This License applies to any program or other work which contains a notice placed by the copyright holder saying it may be distributed under the terms of this General Public License. The "Program", below, refers to any such program or work, and a "work based on the Program" means either the Program or any derivative work under copyright law: that is to say, a work containing the Program or a portion of it, either verbatim or with modifications and/or translated into another language. (Hereinafter, translation is included without limitation in the term "modification".) Each licensee is addressed as "you".

   Activities other than copying, distribution and modification are not covered by this License; they are outside its scope. The act of running the Program is not restricted, and the output from the Program is covered only if its contents constitute a work based on the Program (independent of having been made by running the Program). Whether that is true depends on what the Program does.

2. You may copy and distribute verbatim copies of the Program's source code as you receive it, in any medium, provided that you conspicuously and appropriately publish on each copy an appropriate copyright notice and disclaimer of warranty; keep intact all the notices that refer to this License and to the absence of any warranty; and give any other recipients of the Program a copy of this License along with the Program.

   You may charge a fee for the physical act of transferring a copy, and you may at your option offer warranty protection in exchange for a fee.

3. You may modify your copy or copies of the Program or any portion of it, thus forming a work based on the Program, and copy and distribute such modifications or work under the terms of Section 1 above, provided that you also meet all of these conditions:

   a. You must cause the modified files to carry prominent notices stating that you changed the files and the date of any change.

   b. You must cause any work that you distribute or publish, that in whole or in part contains or is derived from the Program or any part thereof, to be licensed as a whole at no charge to all third parties under the terms of this License.

   c. If the modified program normally reads commands interactively when run, you must cause it, when started running for such interactive use in the most ordinary way, to print or display an announcement including an appropriate copyright notice and a notice that there is no warranty (or else, saying that you provide a warranty) and that users may redistribute the program under these conditions, and telling the user how to view a copy of this License. (Exception: if the Program itself is interactive but does not normally print such an announcement, your work based on the Program is not required to print an announcement.)

   These requirements apply to the modified work as a whole. If identifiable sections of that work are not derived from the Program, and can be reasonably considered independent and separate works in themselves, then this License, and its terms, do not apply to those sections when you distribute them as separate works. But when you distribute the same sections as part of a whole which is a work based on the Program, the distribution of the whole must be on the terms of this License, whose permissions for other licensees extend to the entire whole, and thus to each and every part regardless of who wrote it.

Thus, it is not the intent of this section to claim rights or contest your rights to work written entirely by you; rather, the intent is to exercise the right to control the distribution of derivative or collective works based on the Program.

In addition, mere aggregation of another work not based on the Program with the Program (or with a work based on the Program) on a volume of a storage or distribution medium does not bring the other work under the scope of this License.

4. You may copy and distribute the Program (or a work based on it, under Section 2) in object code or executable form under the terms of Sections 1 and 2 above provided that you also do one of the following:

   a. Accompany it with the complete corresponding machine-readable source code, which must be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   b. Accompany it with a written offer, valid for at least three years, to give any third party, for a charge no more than your cost of physically performing source distribution, a complete machine-readable copy of the corresponding source code, to be distributed under the terms of Sections 1 and 2 above on a medium customarily used for software interchange; or,

   c. Accompany it with the information you received as to the offer to distribute corresponding source code. (This alternative is allowed only for noncommercial distribution and only if you received the program in object code or executable form with such an offer, in accord with Subsection b above.)

   The source code for a work means the preferred form of the work for making modifications to it. For an executable work, complete source code means all the source code for all modules it contains, plus any associated interface definition files, plus the scripts used to control compilation and installation of the executable. However, as a special exception, the source code distributed need not include anything that is normally distributed (in either source or binary form) with the major components (compiler, kernel, and so on) of the operating system on which the executable runs, unless that component itself accompanies the executable.

   If distribution of executable or object code is made by offering access to copy from a designated place, then offering equivalent access to copy the source code from the same place counts as distribution of the source code, even though third parties are not compelled to copy the source along with the object code.

5. You may not copy, modify, sublicense, or distribute the Program except as expressly provided under this License. Any attempt otherwise to copy, modify, sublicense or distribute the Program 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.

6. You are not required to accept this License, since you have not signed it. However, nothing else grants you permission to modify or distribute the Program or its derivative works. These actions are prohibited by law if you do not accept this License. Therefore, by modifying or distributing the Program (or any work based on the Program), you

indicate your acceptance of this License to do so, and all its terms and conditions for copying, distributing or modifying the Program or works based on it.

7. Each time you redistribute the Program (or any work based on the Program), the recipient automatically receives a license from the original licensor to copy, distribute or modify the Program subject to these terms and conditions. You may not impose any further restrictions on the recipients' exercise of the rights granted herein. You are not responsible for enforcing compliance by third parties to this License.
8. If, as a consequence of a court judgment or allegation of patent infringement or for any other reason (not limited to patent issues), conditions are imposed on you (whether by court order, agreement or otherwise) that contradict the conditions of this License, they do not excuse you from the conditions of this License. If you cannot distribute so as to satisfy simultaneously your obligations under this License and any other pertinent obligations, then as a consequence you may not distribute the Program at all. For example, if a patent license would not permit royalty-free redistribution of the Program by all those who receive copies directly or indirectly through you, then the only way you could satisfy both it and this License would be to refrain entirely from distribution of the Program.

   If any portion of this section is held invalid or unenforceable under any particular circumstance, the balance of the section is intended to apply and the section as a whole is intended to apply in other circumstances.

   It is not the purpose of this section to induce you to infringe any patents or other property right claims or to contest validity of any such claims; this section has the sole purpose of protecting the integrity of the free software distribution system, which is implemented by public license practices. Many people have made generous contributions to the wide range of software distributed through that system in reliance on consistent application of that system; it is up to the author/donor to decide if he or she is willing to distribute software through any other system and a licensee cannot impose that choice.

   This section is intended to make thoroughly clear what is believed to be a consequence of the rest of this License.
9. If the distribution and/or use of the Program is restricted in certain countries either by patents or by copyrighted interfaces, the original copyright holder who places the Program under this License may add an explicit geographical distribution limitation excluding those countries, so that distribution is permitted only in or among countries not thus excluded. In such case, this License incorporates the limitation as if written in the body of this License.
10. The Free Software Foundation may publish revised and/or new versions of the General Public 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.

    Each version is given a distinguishing version number. If the Program specifies a version number of this License which applies to it and "any later version", you have the option of following the terms and conditions either of that version or of any later version published by the Free Software Foundation. If the Program does not specify a version number of this License, you may choose any version ever published by the Free Software Foundation.

11. If you wish to incorporate parts of the Program into other free programs whose distribution conditions are different, write to the author to ask for permission. For software which is copyrighted by the Free Software Foundation, write to the Free Software Foundation; we sometimes make exceptions for this. Our decision will be guided by the two goals of preserving the free status of all derivatives of our free software and of promoting the sharing and reuse of software generally.

NO WARRANTY

12. BECAUSE THE PROGRAM IS LICENSED FREE OF CHARGE, THERE IS NO WARRANTY FOR THE PROGRAM, TO THE EXTENT PERMITTED BY APPLICABLE LAW. EXCEPT WHEN OTHERWISE STATED IN WRITING THE COPYRIGHT HOLDERS AND/OR OTHER PARTIES PROVIDE THE PROGRAM "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESSED OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. THE ENTIRE RISK AS TO THE QUALITY AND PERFORMANCE OF THE PROGRAM IS WITH YOU. SHOULD THE PROGRAM PROVE DEFECTIVE, YOU ASSUME THE COST OF ALL NECESSARY SERVICING, REPAIR OR CORRECTION.
13. IN NO EVENT UNLESS REQUIRED BY APPLICABLE LAW OR AGREED TO IN WRITING WILL ANY COPYRIGHT HOLDER, OR ANY OTHER PARTY WHO MAY MODIFY AND/OR REDISTRIBUTE THE PROGRAM AS PERMITTED ABOVE, BE LIABLE TO YOU FOR DAMAGES, INCLUDING ANY GENERAL, SPECIAL, INCIDENTAL OR CONSEQUENTIAL DAMAGES ARISING OUT OF THE USE OR INABILITY TO USE THE PROGRAM (INCLUDING BUT NOT LIMITED TO LOSS OF DATA OR DATA BEING RENDERED INACCURATE OR LOSSES SUSTAINED BY YOU OR THIRD PARTIES OR A FAILURE OF THE PROGRAM TO OPERATE WITH ANY OTHER PROGRAMS), EVEN IF SUCH HOLDER OR OTHER PARTY HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

END OF TERMS AND CONDITIONS

How to Apply These Terms to Your New Programs

If you develop a new program, and you want it to be of the greatest possible use to the public, the best way to achieve this is to make it free software which everyone can redistribute and change under these terms.

To do so, attach the following notices to the program. It is safest to attach them to the start of each source file to most effectively convey the exclusion of warranty; and each file should have at least the "copyright" line and a pointer to where the full notice is found.

```
one line to give the program's name and an idea of what it does.
Copyright (C) 19yy  name of author

This program is free software; you can redistribute it and/or
modify it under the terms of the GNU General Public License
as published by the Free Software Foundation; either version 2
of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful,
but WITHOUT ANY WARRANTY; without even the implied warranty of
MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
GNU General Public License for more details.

You should have received a copy of the GNU General Public License
along with this program; if not, write to the Free Software
Foundation, Inc., 675 Mass Ave, Cambridge, MA 02139, USA.
```

Also add information on how to contact you by electronic and paper mail.

If the program is interactive, make it output a short notice like this when it starts in an interactive mode:

```
Gnomovision version 69, Copyright (C) 19yy name of author
Gnomovision comes with ABSOLUTELY NO WARRANTY; for details
type 'show w'.  This is free software, and you are welcome
to redistribute it under certain conditions; type 'show c'
for details.
```

The hypothetical commands '`show w`' and '`show c`' should show the appropriate parts of the General Public License. Of course, the commands you use may be called something other than '`show w`' and '`show c`'; they could even be mouse-clicks or menu items—whatever suits your program.

You should also get your employer (if you work as a programmer) or your school, if any, to sign a "copyright disclaimer" for the program, if necessary. Here is a sample; alter the names:

```
Yoyodyne, Inc., hereby disclaims all copyright
interest in the program 'Gnomovision'
(which makes passes at compilers) written
by James Hacker.

signature of Ty Coon, 1 April 1989
Ty Coon, President of Vice
```

This General Public License does not permit incorporating your program into proprietary programs. If your program is a subroutine library, you may consider it more useful to permit linking proprietary applications with the library. If this is what you want to do, use the GNU Library General Public License instead of this License.

Function Index

*

A

Concept Index

A

B

C

D

E

F

G

H

I

J

L

M

N

O

P

Q

R

S

T

U

V

W