CLU User's Guide

Dorothy Curtis and Stephen Garland
MIT Laboratory for Computer Science

January 22, 1993

This guide describes how to use the portable compiler and debugger for CLU. This compiler and debugger make CLU available on a wider variety of architectures than the earlier native CLU compilers that ran only on Vaxes and 6800's. The CLU language is described in the book *Abstraction and Specification in Program Development* by Barbara Liskov and John Guttag (MIT Press, 1986), and also in the *CLU Reference Manual* by Liskov *et al* (Springer-Verlag, 1981).

1 The CLU compiler

A CLU program consists of one or more modules, also known as *abstractions*. The *interface specification* for an abstraction completely describes how clients (i.e., other abstractions) see the abstraction and how they can use it. The interface specification for a procedural or iteration abstraction is determined by the header for that procedure or iterator; the interface specification for a data abstraction (i.e., a cluster) is determined by the header for the cluster together with the headers for the operations named in the cluster header. The implementation of each abstraction is theoretically invisible to its clients.

Generally, the code for each module is kept in a separate file, the name of which ends with *.clu*. Sometimes it is convenient for several modules to employ common declarations for compile-time constants, e.g., $maxLen = 100$ or $intSeq = seq[int]$. Such "equates" are generally kept in a separate file, the name of which ends with *.equ*.

1.1 Setting up your environment

To make it easy to use CLU, you should customize your Unix environment as follows. Let *~CLU* be the directory in which CLU is installed. This directory may be */usr/clu*, */usr/pclu*, */usr/local/lib/clu*, or some other directory. Check with your system administrator to find where *~CLU* is.

- Put the command `setenv CLUHOME clu-location` in your *.cshrc* file, where *~CLU* is located at `clu-location`. This enables various parts of the CLU system to find the information they need in *~CLU*.
- Put `$CLUHOME/exe` on your Unix search path, e.g., by adding it to the `set path` command in your *.cshrc* file. This allows you to invoke the CLU compiler (*pclu*) and the CLU indenter (*cludent*).

1.2 Compiler commands

The CLU compiler compiles *.clu* files into *.c* files, which must then be compiled and linked using the C compiler.[1] The CLU compiler can be invoked from the Unix shell by the command `pclu`. Useful compiler commands include the following.

Command	Effect
`ce`	create a compilation environment
`xce`	extend a compilation environment
`spec`	add interface specifications to the type library (ignoring implementations)
`check`	check syntax and semantics without generating a *.c* file
`compile`	check and generate a *.c* file (if there are no errors)
`dump`	dump the type library to a *.lib* file
`merge`	read a type library from a *.lib* file
`optimize`	turn on optimization
`optimize false`	turn off optimization
`ext false`	turn off reporting of external abstractions
`help`	provide a summary of all compiler commands
`quit`	exit from the compiler

[1] Earlier native CLU compilers for Vaxes and 6800's compiled *.clu* into object *.bin* files, which were linked by a CLU linker. The portable compiler compiles into C rather than into machine language.

1.3 How the compiler works

The CLU compiler maintains a type library and a compilation environment (CE). The type library contains specifications of abstractions; initially, it contains the specifications for CLU's built-in modules. The CE contains "equate" information; it is initially empty. In order to do proper type checking between abstractions, the CLU compiler adds interface specifications to its type library. When it processes an abstraction, the compiler remembers the interface of the abstraction in the type library and uses it in future type checking.

When an abstraction is successfully compiled, two messages may follow the compilation. The first lists undefined abstractions, that is, abstractions that do not have specifications in the type library. Undefined abstractions should be avoided, because they make it impossible for the compiler to perform complete type-checking. The second message lists abstractions used in the file being processed whose specifications are known. (This message can be suppressed by issuing the command `ext false` to the compiler.)

There are three commonly used ways to get specifications into the type library.

1. `merge` in an appropriate type library (*.lib* file).
2. `spec`, `check`, or `compile` the *.clu* file containing the abstraction.
3. `spec` a *.spc* file containing the interface specification for the abstraction. (A *.spc* file differs from a *.clu* file in that it contains procedure and cluster headers, but no implementations.)

The top most procedure must be named `start_up`.

1.4 Command Descriptions

Following are descriptions of the most commonly used compiler commands. Each command can be abbreviated to two or more characters (e.g., `compile` can be abbreviated to `co`).

- `ce` *filename* { , *filename* ... }

  The `ce` command creates a compilation environment from the named *.equ* files. Any previously existing compilation environment is forgotten. (See `xce` comand.) The compiler will supply the suffix *.equ* if it is not part of *filename*. Each file named in this command must contain only equates.

- `xce` *filename* { , *filename* ... }

  The `xce` command adds the equates in the named *.equ* files to the current compilation environment.

- `check` *filename* { , *filename* ... }

  The `check` command checks for syntax and type errors in the named *.clu* files. The compiler will supply the suffix *.clu* if it is not part of *filename*.

- `compile` *filename* { , *filename* ... }

  The `compile` command compiles the named *.clu* files into *.o* files. The compiler will supply the suffix *.clu* if it is not part of *filename*. A *.o* file is produced only if there are no errors. Hence, if a *.o* file is produced, any messages are warnings, not errors. The compiler generally produces *.o* files that are suitable for use with the CLU debugger. Once a program has been debugged, more efficient *.o* files can be produced by issuing the `opt` command to the compiler before the `compile` command.

- `spec` *filename* { , *filename* ... }

  The `spec` command enters the interfaces of the abstractions in the named *.spc* or *.clu* files into the type library without type-checking any implementation bodies. The compiler will supply the suffix *.spc* or *.clu* if it is not part of *filename*.

  The type library initially contains the interfaces for the modules defined in the CLU reference manual. Every module that is processed successfully with a `command`, `check`, or `spec` command has its interface added to the library. However, the compiler makes only a single pass over a file, so references within one module in a file to a module defined later in the file will not be type-checked unless the interface of the referenced module is already in the library. To get complete type-checking, it is generally necessary to `spec` or `check` all modules/files in a program before compiling them.

- `dump` *filename*

  The compiler retains in the type library the interface of every module it sees. This library can be dumped to a *.lib* file with the `dump` command and later reloaded with the `merge` command. The compiler will supply the suffix *.lib* if it is not part of *filename*. Libraries make complete type-checking much easier and faster, particularly for very large programs.

- `merge` *filename* { , *filename*, ... }

  The `merge` command loads libraries from the given *.lib* files and merges them with the current type library. The compiler will supply the suffix *.lib* if it is not part of *filename*. If the current library already contains an interface for a module in the library being loaded, the new interface will replace the old.

- `optimize` [`false`]

  The `optimize` command turns code optimization on or off. Optimized code runs faster. Unoptimized code provides more information when used with the CLU debugger.

1.5 Producing an executable program

After you have successfully compiled your *.clu* files into *.c* files, you must compile and link these files using the C compiler. The easiest way to do this is to use the Unix `make` facility, which can also be used to compile your *.clu* files. To use `make`, you generally construct a file named *Makefile* and type the shell command `make`. To use a makefile named *myMakefile*, type `make -f myMakefile`.

Figure 1 contains a sample makefile that can be used to compile the file named *factorial.clu* into a program that can be used with the CLU debugger.[2]

The makefile specifies that the executable file *factorial* depends on the object file *factorial.o*,, which in turn depends on the CLU source file *factorial.clu*. The makefile performs the following actions when it is invoked.

1. If *factorial.clu* is newer than *factorial.o*, or if *factorial.o* does not exist, then the makefile invokes the CLU compiler with the following commands.

Command	Effect
`ext false`	turns off reporting of external abstractions
`spec factorial`	enters interface specifications in type library
`compile factorial`	compiles *factorial.clu* into *factorial.c*

   It is important to `spec` the source file before compiling it, so that specifications for abstractions that occur later in the file (e.g., a procedure named *factorial*) can be used to type-check routines that occur earlier (e.g., the *start_up* procedure).

[2]The three indented lines in the makefile begin with a `tab` character. The `make` program will not work if these `tab` characters are replaced by spaces.

```
# Sample CLU makefile

CLU     = ${CLUHOME}/exe/pclu                          # CLU compiler
LDFLAGS = -L${CLUHOME}/code -lpclu_debug -lgc -lm  # C linker flags
DEBUG   = ${CLUHOME}/debug/*.o                         # debugger support

factorial: factorial.o
        ${CC} -o factorial factorial.o ${DEBUG} ${LDFLAGS}

factorial.o: factorial.clu
        rm -f factorial.c
        ${CLU} -ext false -spec factorial -compile factorial

clean:
        rm -f factorial factorial.o factorial.c
```

Figure 1: Sample makefile for creating programs for CLU debugger

2. If *factorial* is newer than *factorial.o*, or if *factorial* does not exist, then the makefile invokes the C linker with the command

```
cc -o factorial factorial.o ${CLUHOME}/debug/*.o \
        -L${CLUHOME}/code -lpclu -lgc -lm}
```

   to link *factorial.o* with the CLU debugger and library files to produce an executable file named *factorial*.

Because the above commands are tedious and difficult to type correctly, programmers are strongly encouraged to put them into makefiles and not to type them by hand.

Once the program *factorial* has been debugged, the makefile in Figure 2 can be used to produce an optimized version of the program. Before typing `make` with this makefile, you must type `make clean` to remove the files created for use with the debugger.

1.6 Managing larger programs

The two makefiles in the previous section are appropriate for small programs. For larger programs, more sophisticated makefiles can save time in several ways.

```
# Sample CLU makefile

CLU     = ${CLUHOME}/exe/pclu                    # CLU compiler
LDFLAGS = -L${CLUHOME}/code -lpclu -lgc -lm      # C linker flags

factorial: factorial.o
        ${CC} -o factorial factorial.o ${LDFLAGS}

factorial.o: factorial.clu
        rm -f factorial.c
        ${CLU} -ext false -spec factorial -opt -comp factorial

clean:
        rm -f factorial factorial.o factorial.c
```

Figure 2: Sample makefile for creating optimized programs

- They can be used to avoid recompiling all modules whenever a single module changes.
- They can shorten the time needed to create a type library by using the `merge` command instead of the `spec` command.

Figure 3 contains a sample makefile for managing a moderate size program. Definitions at the beginning of the makefile describe where to find the modules that must be compiled and linked to build an executable program:

- `SOURCES` contains a list of the four source *.clu* files for the program.
- `EQUATES` is the name of an *.equ* file containing equates (such as $strSeq = sequence[string]$) that are used in several of the source files.
- `LIBRARY` is the name of a CLU type library that will be constructed from the interface specifications in the source files.
- `OBJECTS` contains a list of the four *.o* files that will be produced from the *.c* files by the C compiler.
- `PROGRAM` is the name of the executable program that will be produced from the *.o* files and the CLU libraries by the C linker.

```
### Sample Makefile for multimodule program

CLU     = ${CLUHOME}/exe/pclu                           # CLU compiler
LDFLAGS = -L${CLUHOME}/code -lpclu_debug -lgc -lm  # C linker flags
CLULIBS = ${CLUHOME}/lib/*.lib                          # CLU libraries
DEBUG   = ${CLUHOME}/debug/*.o                          # debugger support

EQUATES = Equates.equ
SOURCES = collection.clu countWords.clu getWord.clu lowercase.clu
LIBRARY = countWords.lib
OBJECTS = collection.o countWords.o getWord.o lowercase.o
PROGRAM = countWords

.SUFFIXES: .o .clu


.clu.o:
        rm -f $*.c
        ${CLU} -ext false -merge ${LIBRARY} -ce ${EQUATES} \
               -compile $<

${PROGRAM}: ${OBJECTS}
        ${CC} -o ${PROGRAM} ${OBJECTS} ${DEBUG} ${LDFLAGS}

${LIBRARY}: ${EQUATES} ${SOURCES}
        make library

library:
        ${CLU} -ext false -merge ${CLULIBS} -ce ${EQUATES} \
               -spec ${SOURCES} -dump ${LIBRARY}

all: ${LIBRARY} ${PROGRAM}

clean:
        rm -f ${LIBRARY} ${OBJECTS} ${PROGRAM}
```

Figure 3: Makefile for multimodule program

The `.SUFFIXES:` line in the makefile declares that *.o* files can be built from *.clu* files, and the three lines beginning with the one that contains `.clu.o:` instruct the makefile to do this using the following commands:

Command	Effect
`ext false`	turns off reporting of external abstractions
`merge ${LIBRARY}`	creates a type library from *countWords.lib*
`ce ${EQUATES}`	creates a compilation environment from *Equates.equ*
`compile $<`	compiles the *.clu* file into a *.o* file

The remainder of the makefile describes the actions it will take in response to user commands.

- The first use of the makefile should be via the command `make all`, which causes the makefile to build the type library *countWords.lib* and then to build the executable program *countWords*.

- Afterwards, whenever there is a change in the implementation of a procedure in one of the source *.clu* files, the command `make` (or `make countWords`) should be typed to rebuild the executable program *countWords*. The makefile will determine automatically which *.o* files are out of date with respect to the recently changed *.clu* files, and it will recompile only those files before relinking *countWords*. It is precisely this behavior of the `make` facility that makes it so attractive: when you change a small part of a large system, `make` notices exactly which files you've changed, and it updates just that the part of the system that depends on those files.

- Whenever there is a change to a procedure or cluster interface in one of the source *.clu* files, the command `make all` must reissued to remake the type library before remaking the executable program.

There are two important limitations of makefiles such as this:

1. Because *.clu* files contain implementations for procedures in addition to headers for clusters and procedures, there is no way for the makefile to determine when a change in a *.clu* file affects an interface and when it affects only the implementation of an interface. Hence, to avoid remaking the *.lib* interface library every time an implementation changes, the makefile requires an explicit `make library` command whenever an interface changes.

2. The makefile starts a separate CLU compilation to remake each out-of-date *.o* file. More sophisticated makefiles can be constructed that start a single compilation to remake all out-of-date *.o* files; these makefiles save time because they read *.lib* type libraries and build compilation environments from *.equ* files just once, rather than several times. But they are much more complicated than the makefile shown here, and are not necessary for moderate-sized projects.

2 Using Emacs with CLU

The *emacs* editor provides convenient facilities for preparing and compiling CLU source files. A special *clu-mode* allows you to access information about CLU using the *emacs* help facility, and it allows you to run the CLU compiler and indenter under *emacs*. It also automatically indents lines in a CLU program as you enter them.

In order to use *emacs* with CLU, you should edit your *.emacs* file (if you do not have one, create one) to load a library that defines a *clu-mode*. You can do this by inserting the file *~CLU/emacs/.emacs* into your *.emacs* file.

2.1 Using Emacs to prepare CLU programs

The *emacs* editor should go into *clu-mode* automatically when you visit any file with a name that ends in *.clu*. Once in *clu-mode*, the following commands are available:

`<ESC> &` Runs the CLU compiler.

When typed in a buffer named *test.clu*, `<ESC> &` causes *emacs* to propose running the CLU compiler with the command `pclu test.cu`. You can edit this command line, or you can replace it with the command `make`, before typing `<RETURN>`; *emacs* will then run `pclu` or `make` and display the results in a separate compilation buffer. You can also edit the command line to `pclu` alone, and then enter compiler commands by positioning the cursor at the compiler prompt in the compilation buffer.

`C-X C-K` Kills a running CLU compiler.

`` C-X ` `` Locates errors after a CLU compile.

If there were errors in a CLU compilation, `C-X `` will position the first error message at the top of the compiler window and put the cursor at the beginning of the line containing the error. Each successive `C-X `` will move to the next error.

`<ESC> <TAB>` Indents the current buffer.

This command indents the file in a more CLU-like fashion than is done automatically in *clu-mode*.

At any time while in *emacs*, you can return the display back to one window by entering either `C-X 1` or `C-X 0`. `C-X 1` returns the entire display to the window in which the cursor is presently in. `C-X 0` returns the entire display to the other window. `C-X b` can be used to move to any buffer.

2.2 CLU help from Emacs

You can use the *emacs* help facility to view information in the CLU reference manual. To get CLU help, type `C-h` (the help key) followed by one of the keys:

- `d` for help on CLU symbols such as `int$mul`, `array`, `ARRAY$LOW`, `FILE_NAME`, `string$s2ac`, `stream$primary_output`.
- `g` for general CLU information on topics such as `terminal_i/o` , `file`, `own_variables`, `CLUSTERS`, `syntactic_sugar`.
- `/` for CLU apropos. Give a partial name, like `file`, and get a list of all the CLU symbols and topics that include that name.

For example, if you type `C-h d`, *emacs* will prompt you with `Describe CLU symbol:` If you type `oneof$make_ <RETURN>`, *emacs* splits your screen and gives help about the CLU operation *oneof$make_*.

You can request information using uppercase, lowercase, or a mixture of uppercase and lowercase. Symbol completion works (hit the `<TAB>` key); for example, if you type `array$` followed by `<TAB>`, you'll get a list of possible completions (all the operations on the CLU *array* abstraction in this case). If the cursor is placed on a CLU symbol when you type `C-h d`, *emacs* will propose giving help about that symbol; for example, if your cursor is placed someplace on `array[foo]$low` in your file, and you type `C-h d`, *emacs* will prompt you with

`Describe CLU symbol (default array$low):` ; typing `<RETURN>` selects this default help topic.

To be reminded of this information, you can type `M-x clu-doc-help` (i.e., `META-x clu-doc-help`).

3 The debugger

The CLU debugger enables users to monitor execution of a program. Users can single step a program, executing one line at a time, or they can set *breakpoints* in the program. The debugger suspends execution of a program whenever it reaches a breakpoint, so that the user can determine how control reached that breakpoint and/or examine the values of program variables.

Whenever a procedure is invoked, information about the invocation is pushed onto a runtime *stack*. This information is known as a *stack frame*. By examining the stack when a breakpoint is reached, the user can determine how control reached the current procedure.

The debugger enables users to print the values of program variables, which are CLU objects. Users can control how much of the representations for objects are displayed. The printing *depth* is the maximum number of levels of nesting of an object that will be displayed. The printing *width* is the maximum number of components of an object that are displayed at each depth. The greater the width, the more elements in an array and the more fields in a record that are printed. The greater the depth, the more information about each element in an array or each field in a record that is displayed.

3.1 Command overview

A debugger command consists of a command name followed by a sequence of arguments separated by spaces. The following describes related groups of debugger commands. The next section contains an alphabetical list of all debugger commands and describes them in detail.

- **Running**

  The `run` and `continue` commands cause the program to run continuously. The `next` command causes execution of a single program line.

- **Breakpoints**

  The `break` command sets breakpoints. The `show` command shows the current breakpoints. The `delete` command deletes breakpoints. The `step` command turns on single stepping for a procedure. The `unstep` command turns off single stepping for a procedure. The `trace` command turns on tracing for exceptions. The `untrace` command turns off tracing for exceptions.

- **The stack**

  The `where` command displays the stack. The `up` and `down` commands select different frames in the stack.

- **Printing**

  The `print` command displays the values of variables. The `width` and `depth` commands control how much of the value is displayed.

- **Invocation**

  The `eval` command evaluates procedure invocations.

- **Source code**

  The `list` command displays source code. The `func` command selects a procedure to be displayed. Reaching a breakpoint also selects the procedure containing as the one to be displayed. Moving up and down the stack changes the procedure to be displayed.

- **Miscellaneous**

  The `help` command gives brief information on debugger commands. All commands can be displayed via `help all`. The `quit` command terminates the debugger.

3.2 Commands

This section describes each command in detail. Commands are listed alphabetically. Users can repeat execution of the `continue`, `eval`, `up`, `list`, `down`, `next` commands simply by pressing the `<RETURN>` key.

- `break` {*breakpoint ...*}

  The `break` command instructs the debugger to stop when a particular procedure or iterator is entered or exited or before a particular line in a file is executed.

Each *breakpoint* is either the name of a procedure, the name of an iterator, or a line number, which refers to a line in the file containing the current procedure (or iterator). Note that the `func`, `up`, and `down` commands change the current procedure. Some examples:

```
break start_up         % sets breakpoints on entry to and
                       % exit from the top-most procedure

break 10               % sets a breakpoint at line 10
                       % in the current procedure

break 10 20            % sets breakpoints at lines 10 and 20
                       % in the current procedure

break t$op1 t$op2      % sets breakpoints on entry to and
                       % exit from procedures t$op1 and t$op2

break t$op1 t$op2 10   % sets breakpoints on entry to and
                       % exit from procedures t$op1 and t$op2
                       % and at line 10 in the current procedure
                       % (which may not be either t$op1 or t$op2)

func t$op1             % select function t$op1
break 10               % sets a breakpoint at line 10 in t$op1
```

Whenever the debugger reaches a breakpoint, it displays the corresponding line in the source file and stops before executing that line.

- `continue`

  The `continue` command directs the debugger to continue running the program, either until the next breakpoint or until the next instruction in a procedure (or iterator) that is being single-stepped. It terminates any single-stepping set by the `next` command in the current procedure (or iterator).

- `delete` *breakpoint* {*breakpoint* ...}

  The `delete` command removes the specified breakpoints. (Note: If there are two breakpoints with the same line number, only the first will be deleted.) The argument `all` causes all breakpoints to be removed. Some examples:

```
delete t$op1                % deletes the breakpoints on entry to
                            % and exit from the procedure t$op1

delete all                  % deletes all breakpoints
```

- `depth` {*integer*}

  The `depth` command controls the printing depth for objects printed by the `print` command. If the `depth` command has no argument, the debugger prints the current depth. See also the `width` command.

- `down` {*integer*}

  The `down` command moves down the specified number of stack frames. If no argument is specified, then 1 is assumed. See also the `where` command and the `up` command.

- `eval` *expression*

  The `eval` command evaluates the specified expression. The following section describes the legal expressions. Some examples:

```
eval a = "xyz"                      % sets the variable a to "xyz"
eval po = stream$primary_output()   % sets po to
                                    %  the primary output stream
eval stream$putl(po, a)             % prints a on po
```

  The assignment to the variable `a` in the example above changes the value of the variable `a` in the current stack frame, if such a variable exists. Otherwise, it creates a debugger variable named `a` and assigns the value of the expression to that variable.

- `func` {*name*}

  The `func` specifies the named procedure (or iterator) to be the one of current interest, either for listing source code or for setting breakpoints at line numbers. If no argument is given, the current line is printed, along with the five preceding and five succeeding lines

```
func t$op1                    % select function t$op1
list                          % list lines surrounding t$op1
break 10                      % set a breakpoint at line 10 in
                              % the file containing t$op1
```

- `help` {*command*}

  The `help` command, with no arguments, gives a summary of command names. When a command name is present as an argument, the `help` command will give a short description of the specified command. If the argument is the keyword `all`, short descriptions of all commands are displayed.

- `list` {*integer ...*}

  The `list` command displays source code in the current procedure (or iterator). By default, the current procedure is the routine that last reached a breakpoint. When the debugger first starts up, the `start_up` procedure is selected as the current procedure. The `func`, `up`, and `down` commands change the current procedure to the specified procedure. The arguments to `list` specify the line numbers of interest (use a space to separate the line numbers). The `where` command can be used to show the current line numbers in the currently active procedures.

  ```
  list                         % list lines at current breakpoint
  func t$op1                   % select function t$op1
  list                         % list lines surrounding t$op1
  ```

- `next` {*integer*}

  The `next` command directs the debugger to single step to the next line in the current procedure. When a numeric argument n is present, the debugger executes the next n lines in the current procedure and prints out the corresponding source code as each line is executed.

- `print` *variable*

  The `print` command prints the values of variables, procedure arguments, and **own** variables. Uninitialized variables are printed as `???`.

- `quit`

  The `quit` command causes the debugger to terminate.

- `rebuild`

  The `rebuild` command causes the debugger to run `make` and then the program.

- `restart`

  The `restart` command causes the debugger to restart the program: a new process is created, causing **own** variables to be initialized and the stack to be empty.

- `run`

  The `run` command causes the debugger to continue running the program.

- `show`

  The `show` command displays the location of the current breakpoints.

- `step` *name*

  The `step` command turns on single stepping for the specified procedure.

- `trace` *name*

  The `trace` command turns on tracing for the named signal.

  ```
  trace bounds                 % trace the bounds exception
  trace all                    % trace all exceptions
  ```

- `unstep` *name*

  The `unstep` command turns off single stepping for the specified procedure.

- `trace` *name*

  The `untrace` command turns off tracing for the named signal.

  ```
  untrace bounds               % stop tracing the bounds exception
  untrace all                  % do not trace any exceptions
  ```

- `up` {`integer`}

  The `up` command moves up the specified number of stack frames. If no argument is specified, 1 is assumed. See also the `where` command and the `down` command.

- `where` {`integer`}

  The `where` command displays the frames on the stack. A numeric argument n limits the output to the last n frames pushed on the stack.

- `width` {`integer`}

  The `width` command controls the printing width for objects printed by the `print` command. If the `width` command has no argument, the debugger prints the current width. See also the `depth` command.

3.3 Expressions

The `eval` command currently accepts the following expressions:

- Constants:
  - booleans: `0` for false, `1` for true
  - characters: `'a'`
  - strings: `"abc"`
  - integers: `1`
  - reals: `1.2`
- Variables:

  The debugger first looks for a debugger variable with the given name, then for a local variable in the current procedure (or iterator), then for an **own** variable in the current procedure (or iterator) and finally for an **own** variable in the current type.

  The current version of the debugger does not always print the value of the variable that one expects: if two parallel scopes use the same name to denote variables of different types, the first declaration determines the print operation.
- Invocations:

  The programmer may invoke stand-alone procedures and procedures from parameterized or unparameterized clusters. Arguments to procedures can be constants or variables, as described above. The current debugger does not recognize sugars: users must type `eval int$add(1,2)` rather than `eval 1+2`. The debugger does not type-check arguments for procedure invocations; irrecoverable errors may result if arguments do not have the appropriate types.

  Some support has been added to ease access to components of complex objects. Instead of `eval record[f1:int, f2: string]$get_f1(r1)`, the user may type `eval rep$get_f1(r1)`.

3.4 Operational notes

3.4.1 Interrupting the debugger

When a program is running, typing `control-c` (i.e., holding down the control or ctrl key and then pressing the letter `c`), will cause control to be passed back to the debugger. This assumes that `control-c` is your `INTR` (interrupt) character. (See *man (2) stty.*) For `control-c` to take effect, the program must pass through a procedure entry/exit point or a procedure with line breakpoints set in it (these lines do not need to be executed). A program in an infinite loop, which does not call a procedure, will not respond to `control-c`. To kill off such a program, type `control-z` (or your `SUSP` (suspend) character), use the `jobs` shell command to locate the errant program, and type

```
kill -9 %number_associated_with_program
```

You may need to use the `reset` shell command to restore normal tty behavior.

3.4.2 Auxiliary files

When a program is executed under the debugger, an auxiliary file with symbol information is created. If the program is named *ps1*, then the file is named *ps1.sym*.

To preserve break point settings between debugging sessions, the debugger maintains the current breakpoints in a file. If the program is named *ps1*, then the file is named *ps1.bkpts*.

3.4.3 A note on stacks

When an iterator is active, the invoking procedure will appear on the stack twice: once for the invocation and once for the body of the **for** statement. Also, low-level routines that are in-lined may not appear on the stack, so setting breakpoints at such low-level routines may have no effect. Own initialization is not traceable and does not appear on the stack.

3.4.4 Capturing output

The (Unix) `script` command provides a method for capturing output created during a debugging session.

3.4.5 Fatal errors

When running a program, a fatal error may occur and result in a Unix `Segmentation Violation` or `Bus Error`. When this happens, the debugger executes a `where` command to display the stack and then terminates. These errors are frequently caused by uninitialized variables.

If a program accesses uninitialized variables without triggering a fatal error, the state of the debugger may be corrupted.

If a circularly-linked object is printed, the debugger may terminate when the stack fills up.

3.4.6 How Printing Works

When printing an object of type t, the debugger uses `t$print`, if it exists and takes an object of type t (or cvt) as its first argument and a pstream as its second argument. If such an operation does not exist, `rep$print` for type t is used.

4 Line editing

The debugger and compiler support a limited amount of line editing. The following files are used to specify key bindings for line editing: *~/.lineedit.keys* and *~/.inputrc*. These files override the default settings given by the following table.

Key	Action	Alternative key
ctrl-A	move to beginning of the line	ESC
ctrl-B	move back (left) one character	left-arrow on lk-201's
ctrl-D	delete current character	
ctrl-E	move to end of current line	
ctrl-F	move forward (right) one character	right-arrow on lk-201's
ctrl-J	complete entry	
ctrl-K	delete to the end of the line	remove-key on lk-201's
ctrl-M	complete entry	
ctrl-N	move to next history item	down-arrow on lk-201's
ctrl-P	move to previous history item	up-arrow on lk-201's
ctrl-U	delete line	
ctrl-W	delete word	
del	delete previous character	
	move forward one word	next-screen on lk-201's
	move back one word	prev-screen on lk-201's