@type ‹documentation›commonlisp.doc

Tops-20 Common Lisp

Red Pages

Charles L. Hedrick

1984

Swiss Cheese Edition
(very drafty)

Table of Contents

1. Introduction

This document contains implementation-dependent information describing the Common Lisp implementation for the DECSYSTEM-20. In the rest of the manual, I will simply refer to it as "Lisp". Lisp is designed to be used with the Common Lisp Reference Manual, (Guy Steele, Carnegie-Mellon University Computer Science Dept.) This manual documents only the peculiarities of this particular implementation. I hope that there aren't very many of those. Indeed many people will probably never need this document at all.

1.1. How to read this manual

This document is organized into the following chapters:

1 General material about the history and goals of DECSYSTEM-20 Common Lisp, and an overview of its internal organization.

2 A description of the current status of the system, including the facilities that are not yet implemented, and various issues affecting the speed of your program.

3 A description of the major user facilities of the system. This chapter contains a number of sections, each giving a general discussion of some facility. It is intended to cover the same material as the next chapter, namely all of the implementation-defined facilities. However this chapter is organized topically, whereas the next one is organized alphabetically by function. Also, it is at a conceptual level, whereas the next chapter is intended as a reference manual. There is a section at the beginning of this chapter that provides an overview of its organization.

4 This is intended as a complete reference manual for all functions that are extensions or whose definition is implementation-dependent. In those cases where a full description seems to belong in the previous chapter, there is a cross-reference to the appropriate section. This chapter is organized alphabetically by function or variable name.

5 Hints for people who want to import system-dependent Spice Lisp code.

1.2. The genealogy of DECSYSTEM-20 Common Lisp

This Lisp is in fact an implementation of Carnegie-Mellon's Spice Lisp. Spice Lisp was originally intended for a micro-coded machine with bit-mapped screen. However implementations based on it are being done for the DECSYSTEM-20 and VAX. We are attempting to keep the Spice implementations as similar as possible. Here are the pieces of DECSYSTEM-20 Common Lisp. with an indication of which of them came from Spice Lisp:

- Compiler - This will be the Spice compiler, with code generation rewritten to produce code for the DEC-20.

- System code - This is the portion of the runtime system written in Lisp. It includes most of the functions that the user calls. These functions are taken directly from Spice, with minor modifications where the code is representation--dependent.

- Kernel - This is the assembly language portion of the system. It contains low-level functions, mostly things that manipulate internal data representations, e.g. CONS and the garbage collector. Most of these functions implement the basic byte codes of the Spice machine. These are documented in the Internal Design of Spice Lisp (Scott Fahlman et al, Carnegie-Mellon Computer Science Dept.) That document should be regarded as the blue pages for this implementation. In addition, we have added some higher-level functions to the kernel, when it seems that this would help performance noticibly. For example, the interpreter (EVAL) and much of READ and PRINT are hand-coded. In general the assembly language code follows the Spice Lisp code very closely.

The interior design is sort of a cross between the Spice machine and Elisp, the Rutgers extended-addressing version of UCI Lisp. The Elisp manual documents most of the internal data structures in detail. By the final release, we will provide a real blue pages that integrates the information in the Elisp manual and the Spice internals manual, but for the moment, those two documents should allow you to find your way around in the code. Fortunately, the internal data representations used by Spice Lisp and Elisp are surprising similar.

1.3. Design Goals

In evaluating this implementation, you might find it useful to know what goals we had in mind.

- We intend this implementation to stick very close to the Standard. The extensions are largely tools for the implementors, which we have made accessible to users. There

are also a few features added to increase compatibility with the VAX implementation. However our experience with Pascal has lead us to realize how important standards are. I believe that Pascal's greatest weakness is that no interesting program written in it is portable. We are determined that this will not be the case with Common Lisp. [Note that what you have now is an development version, intended primarily for boostrapping the compiler. Thus some pieces of the system are still missing. Rest assured that we will supply all of those missing pieces before releasing this version officially.]

- We are quite concerned about performance. However we are interested in the performance that a normal researcher will see, rather than in providing tools to let benchmarks be tuned to blinding speeds. This means that we worry most about programs that use no declarations and which are written without undue concern for speed.

1.4. Overview of the Design

Lisp uses extended addressing, which gives it a much larger address space than conventional programming languages. Lisp runs only on Model B KL-10 processors running TOPS-20 release 5 or later. That is because extended addressing is only implemented for those systems. In particular, Lisp does not run on TOPS-10, on older 2040's and 2050's (those with Model A CPU's), or on 2020's.

The internal design of Lisp is modelled after the Lisp Machine. All Lisp objects are type-coded pointers. They consist of 3 fields:

- high order bit is used by the garbage collector for marking. It is normally off (for extended addressing to work).

- next 5 bits are a type code, used internally by the system.

- last 30 bits are the data for the object. In most cases this is the address of the object itself. However in some cases the actual object fits in 30 bits, and no pointer is needed. E.g. we have 30-bit integer constants.

There are two free spaces. Most Lisp pointers point to objects within one of these spaces. When a space is full, a copying garbage collector is invoked to copy all currently used objects to a new space.

This implementation is a shallow-binding Lisp. It stores atom bindings in a "value cell" associated with the atom, saving old bindings on a pushdown stack. List cells take two words, each containing one object. Atoms consist of small blocks of memory, with the following structure:

value cell

pointer to property list

string pointer to pname

function definition, or NIL if none

other internal information involving function definitions - set by DEFUN or other function-defining forms, not directly visible to the user

2. Current Status of the System

This is a preliminary release of this system. The basic data structures are in their final form. So is almost all of the kernel code. However a few features are not yet implemented. We also plan to make some additional performance improvements to the system.

There are two serious omissions:

- There is no compiler. For this reason, the Lisp-level system code is running interpretively. This can slow down some kinds of programs quite significantly.

- Irrational functions such as SIN, COS, and SQRT are not implemented. They will eventually be loaded from the Fortran library.

There may be other oversights. If so, we would appreciate having them brought to our attention.

Bugs are documented in the file RUTGERS::T:<SLISP.CODE>BUGS. Please report any errors, even minor ones, that are not in this file.

2.1. Efficiency Issues

The most serious problem is, of course, the lack of a compiler. Since the compiler is written in Common Lisp, we had to bring up an interpreted version before we could do development work on the compiler. What you see now is the bootstrap version intended to support compiler work.

The system code makes heavy use of interpreted macros. Because of the CPU time involved in expanding these systems of interlocking macros, functions tend to run very slow the first time they are called, and then faster on later calls. For complex functions such as the sequence functions, this effect may occur several times, as various options are exercized. Currently Lisp is set up so that it saves the results of macro expansions. So any given macro call only needs to be expanded once. Because of the heavy use of macros by system code, we strongly urge that you leave this feature enabled.

There are still major inefficiencies in the system. We will fix these over the next 6 months or so. These inefficiencies can cause slowdowns ranging from factors of 2 to 50. The most serious slowdowns are in the string and sequence functions. These use DO loops and array indexing. We hope to hand-code them to use small ILDB loops. This will probably not affect most traditional Lisp code, however.

3. User Facilities

This chapter contains the following sections:

3 - how to run the system, and what the top-level is like. A description of the system-wide help convention.

3.2 - the break handler. This is an interactive system which is entered when an error happens.

3.3 - TRACE, a function that you can use to get a trace of your program's behavior.

3.4 - STEP, a function that you can use to control your program's execution on a expression by expression basis, seeing the results of each evaluation.

3.5 - the editor. which is actually an interface to EMACS

3.6 - some miscellaneous facilities primarily for system builders: Customizing the top level (including changing the prompt), creating a saved core image file, loading code into a specified package, and calling DDT.

3.7 - details about the I/O implementation, including how the various OPEN options work, the way the Common Lisp and TOPS-20 file models are matched, end of line handling, and details about how I/O is done to specific devices (particularly terminals). This section has a paragraph at the beginning that describes its organization.

We intend the Lisp system to be installed on your system as SYS:CLISP.EXE. If it is, you start it by typing

```
CLISP
```

Lisp has a simple EVAL top level. You type Lisp forms to it, and it prints the result. If the form returns multiple values, you will see all of them (each on a new line).

? should usually give you useful information about the context you are currently in. In many cases you will have to type a carriage return after the ?. At the top level, it tells you how to define functions, and describes some of the most important facilities. In other situations ? is rebound to messages that are useful in that context. We urge users to continue this convention for packages that they write.

3.1. Interrupt Characters

Several interrupt characters are defined. When you type one of these characters once, its effect will happen the next time the program reads from the terminal. If you type it a second time, the effect will happen immediately. most of the time. If your program happens to be in the middle of a garbage collection, the effect will normally be delayed until the end of the garbage collection.

^B (Break)
: This causes Lisp to enter the break package, just as if an error had happened. This is sometimes useful if you think your program is in an infinite loop. You can use the commands in the break package to look around. Currently there is no way to continue your program after you have done this.

^C
: This will return you to the EXEC. If you are in the garbage collector, it will delay the return until the garbage collection is finished. If you type more than one ^C, Lisp will count them. If you 6 of them, it will return you even if you are in the garbage collector. This is to protect against bugs in the garbage collector that would otherwise make it impossible to escape from Lisp.

^G
: [Note that ^G is the bell.] This will return you to the top level of Lisp. If you are currently in a break loop, it will return you to the top-most break loop.

^Y
: This is a high-priority version of ^C. It always causes Lisp to exit, even if a garbage collection is going on. It takes precedence over any other interrupts that may be in progress. It is intended to make sure that you can always get out of Lisp, even if bugs exist in the ^C code. We may remove ^Y when Lisp is finally released, if ^C seems to be reliable.

3.2. The Break Facility

The default condition handlers for errors call a built-in break package. This is a specialized READ-EVAL-PRINT loop. It evaluates forms in the context of the bad form. You can find the values of special variables by typing their names. The following forms have special actions when typed to the break system, and thus can be thought of as commands to it. ($ can be typed as either an escape or a dollar sign. Escape is normal.)

$G
: Returns you to the top-level loop, i.e. exits the break abruptly.

$P
: Attempts to proceed from the break. This will only work if

the error is "correctable". For this to work, you have to know how to correct the error. Some cases are obvious. If a function is undefined, you must define it. If a variable is unbound you must set it to a value. In fact these are the main cases where $P is useful. Most other error types require you to return a value, which will then be used to repair the error. This requires (RETURN value), which is documented below. $P is equivalent to (RETURN nil).

(RETURN ‹value›)
: Attempt to proceed from the break, returning the specified value. This value is returned to the error handler. It is used in an attempt to repair the error. E.g. if the system complains that something is not an symbol, you should return a symbol. The system will attempt to do whatever it was trying to do, using the symbol you return instead of the original non-symbol.

$S
: Displays the most recent call executed from user code, and then the call stack. Except for the most recent user call, only calls of user functions are shown. (Yes. I know. This facility is a pretty poor excuse for a stack display. The final version will have a more useful debugger.)

?
: Displays this list of commands.

3.3. Trace

The trace facility allows you to ask for printout whenever a certain function is called. The printout shows the arguments with which it is called and the value returned. It is indented to show recursion. Here is a typical example:

```
* (defun fact (n)
        (cond ((zerop n) 1) (t (* n (fact (1- n))))))
FACT
* (trace fact)
FACT
* (fact 4)

0: (FACT 4)
. 1: (FACT 3)
. . 2: (FACT 2)
. . . 3: (FACT 1)
. . . . 4: (FACT 0)
. . . . 4: returned 1
. . . 3: returned 1
. . 2: returned 2
. 1: returned 6
0: returned 2424
*
```

There are a number of options, to allow for more selective output. The full form of TRACE is

```
(TRACE function(s) :CONDITION form :BREAK form
                   :WHEREIN symbol-or-list
                   :ENTRY-PRINT list-of-forms
                   :EXIT-PRINT list-of-forms)
```

You may leave out the keyword parameters if you do not need them. Here is what they do:

function(s)
: A function or list of functions to trace. Note that you should not quote function names.

:CONDITION
: A form that controls whether the trace information is printed. It will be EVAL'ed at each entry to the function.

:BREAK
: A form that controls whether a break will occur before and after the function is executed. It will be EVAL'ed at each entry to the function.

:WHEREIN
: Allows you to specify that tracing should happen only if the function is called inside another specific function. This may be either a symbol or a list of symbols.

:ENTRY-PRINT
: A list of forms to EVAL and PRINT at the start of each call.

:EXIT-PRINT
: A list of forms to EVAL and PRINT at the end of each call.

To turn off tracing, use (UNTRACE). Untrace checks to see that its args are all symbols. If they are, it returns a form which will untrace each one. Otherwise, it signals an error, and none of the forms are untraced. With no args, untraces all traced functions.

3.4. The Stepper

The single stepper is another facility to make it easier to debug functions. It allows you to watch the interpreter EVAL each form individually. Here is an example of what it is supposed to look like [see below for an explanation of why it does not]:

```
* (step (fact 3))
 (FACT 3)n
  3n
  3
  (COND ((ZEROP N) 1) (T (* N (** **))))n
   (ZEROP N)n
    Nn
    3
   NIL
   Tn
   T
   (* N (FACT (1- N)))n
    Nn
    3
    (FACT (1- N))s
    2
   6
  6
 6
6
* ^C
```

I typed the lower-case "n"'s and "s" and pressed return. Notice what it is doing: It types out a form, and then waits for me to type something. If I type N, it evaluates that form and prints the result. If this involves evaluating another form, it stops for that, too. Typing S causes it to evaluate the form without showing what going on inside it.

Here is a complete list of commands to the stepper. If you type "?" while in step mode, you will get this list:

N (next)
    evaluate current expression in step mode.

S (skip)
    evaluate current expression without stepping.

M (macro)
    steps a macroexpansion, signaled by a :: prompt.

Q (quit)
finish evaluation, but turn stepper off.

P (print)
print current exp (ignoring *step-prinlevel* & *step-prinlength*.)

B (break)
enter break loop.

E (eval)
evaluate an arbitrary expression.

? (help)
print this text.

R (return)
prompt for an arbitrary value to return as result of current exp.

G throw to top level.

The stepper automatically refuses to step through system code, even when it is interpreted. If you need to debug system code with the stepper, you should look at the macro STEP-STEP-FORM in STEP.CLISP. This is where system functions are made un-steppable.

3.5. The Editor

Lisp uses EMACS as its editor. You can call it with the function ED, described in the manual, or EDIT. EDIT is just like ED, except it does not evaluate its argument. In most cases, EDIT is probably more convenient. Otherwise these functions are identical.

As described in the Common Lisp manual, there are three different things you can do with EMACS:

(ED symbol)
Edit a function definition. Lisp will pretty-print the current definition into the EMACS buffer and call EMACS. When you are finished editing, type ^X^Z (the normal EMACS command to return to the superior). Lisp will read the first S-expression back in from the EMACS buffer and EVAL it. Should you decide that you don't want to redefine the function, put something innocuous at the beginning of the buffer (e.g. a NIL).

(ED pathname)
Edit a file. Lisp will simply call EMACS and pass it a request to edit the specified file. When you are finished editing, type ^X^Z to return to Lisp. Lisp will not do anything additional. If you want to write out the modified

file, do ^X^S (or your favorite file-saving command) before exiting. If you want to read in the file after modifying it. you can use the LOAD command.

(ED)

With no arguments, ED simply reenters EMACS. Whatever you edited last is still there. ^X^Z will return to Lisp. Lisp will not do anything additional. such as reading in from the buffer.

I am well aware that this interface leaves much to be desired. The primitives are present in Lisp to do as hairy an interface to EMACS as you like. We are planning an interface modelled after the Maclisp LEDIT.

There is also a function (KILL-EDITOR). It kills the EMACS fork.

3.6. Special features for system builders

This section documents some internals of Lisp that you may find useful if you are building a system of your own.

(%TOP-LEVEL) - never returns

When a copy of Lisp is started, it first prints out the greeting message (set by SAVE - see below) and then calls LISP::%TOP-LEVEL. LISP::%TOP-LEVEL should be a function of no arguments that never returns. If you redefined LISP::%TOP-LEVEL, the redefinition should not take effect until a saved core image is run. The current incarnation will not be affected, since Lisp has already started the existing top level function, and it will never return.

If you intend to use the error handlers that we supply, your top level function should include (CATCH 'LISP::TOP-LEVEL-CATCHER ...) around any EVAL's. That is because the $G function within the error handler THROWS to LISP::TOP-LEVEL-CATCHER.

Should %TOP-LEVEL return, you will be in a READ-EVAL-PRINT loop in the kernel. It prompts with a "*". It is a minimal top-level. intended for testing the kernel.

PROMPT - variable

If you prefer to use the existing top level, you can change its prompt to anything you like. The variable *PROMPT* is PRINC'ed to produce the prompt. It will normally be a string, without any newlines. (FRESH-LINE is called right before printing the prompt.)

(SAVE filename &OPTIONAL greeting-message)

The SAVE function can be used to produce an executable file containing the current Lisp system. The first argument is a

file name, which is passed to OPEN. The second argument (which is optional) is a normally a string. It is PRINC'ed when the saved core image is started. It is intended as a greeting message. If this argument is not supplied, or is NIL, the PRINC is not done.

(LOAD filename :PACKAGE package)

LOAD has an extra option, :PACKAGE. This allows you to specify the package into which the code is to be loaded. The system code must be in the internal Lisp package, not the user's package. So if you wanted to load a new version of PPRINT.CLISP (the pretty-printer), you would type

```
(LOAD "PPRINT.CLISP" :PACKAGE *LISP-PACKAGE*)
(LISP::PPRINT-INIT)
```

(DDT)

(DDT) calls DDT in section 1 (the section in which the kernel code is loaded). It gives DDT access to the kernel's symbol table. To return to Lisp, type

```
RET$X
```

where $ is an escape. Be careful about using $X in DDT to single-step. There is currently a bug in DDT that causes extended-addressing byte instructions to be incorrectly simulated in $X and $$X.

3.7. I/O Implementation

The Common Lisp specificiations leave some aspects of I/O up to the implementor. This section will describe what has been done with some of them. It has the following subsections:

3.7.1 - opening files, including details of filename handling, and how the various OPEN options are implemented.

3.7.2 - how the Common Lisp file model is mapped onto TOPS-20, including file structure, random access, and end of line handling.

3.7.3 - details on how Lisp handles various devices. The most interesting is the terminal. This section describes a number of options you have to control how Lisp interfaces with the terminal.

3.7.1. Opening files

A NAMESTRING is simply a TOPS-20 file specification. Host names go at the beginning of the string, followed by "::". For example "RUTGERS::PS:<HEDRICK>CLISP.EXE". Note however that host names don't have any effect at the moment. The filename parser understands all of the options that TOPS-20 normally understands, including wildcards and the special version numbers 0, -1, -2, and -3.

There may be a slight problem with namestrings because of ambiguity about null file types. In most cases, a field in the file specification can be omitted if it is not specified. Unfortunately, there is no way to omit the file type if the version is specified. "SOURCE..3" is interpreted by TOPS-20 as having a null file type. That is, the file type is specified, and is the null string. If you need to specify the version and leave the file type unspecified, you will simply have to leave the result in pathname format.

All of the keywords described in the manual as "suggested" are implemented except for INSTALLED. If someone can suggest a reasonable meaning for it in TOPS-20, I will be happy to implement it.

Currently Lisp cannot do network I/O. Thus host names are ignored when opening files. The functions that manipulate namestrings and pathnames do handle host names properly. We intend to implement Internet I O eventually.

All of the OPEN options are implemented. Here are some details:

- NEW-VERSION operates according to TOPS-20 conventions. That is, if you specify an explicit version number, that version will be used, and NEW-VERSION will be ignored. This gives an effect similar to SUPERSEDE.

- If you specify UNSIGNED-BYTE or SIGNED-BYTE without a number, you will get 8-bit bytes. UNSIGNED-BYTE allows any byte size up to 35, and SIGNED-BYTE allows any byte size up to 36. Note that you may specify UNSIGNED-BYTE or SIGNED-BYTE even if you intend to use a file for text I/O. This allows you to handle text files with non-standard byte size. For example, if you open a file for (SIGNED-BYTE 8), READ-BYTE will return a signed integer, but READ-CHAR will still return a character. Note that the byte size may affect the way certain devices work. For example, opening a terminal with a byte size of 8 will cause I/O to occur in binary mode.

- DEFAULT gives you STRING-CHAR. STRING-CHAR represents 7-bit ASCII characters. This is the normal Tops-20 representation for text.

- RENAME and RENAME-AND-DELETE rename the file to have a file type of "LISP-BACKUP". If there is more than one version of the file, they are all renamed.

3.7.2. Representation of files and lines

The file model that Common Lisp uses is very close to the DEC-20's actual file model. Thus most I/O is quite straightforward. TOPS-20 files have user-determined byte size. All I/O is done in terms of these bytes. The file length as shown in a VDIRECTORY command gives the number of bytes. This all corresponds nicely to Common Lisp. The Common Lisp OPEN function allows you to specify the byte size to be used for the file. FILE-LENGTH returns the file size in these bytes. NB: FILE-LENGTH will use the byte size that you specified when you OPENed the file. If you are reading an existing file, this might not be the same as the byte size used to write the file. Thus FILE-LENGTH might not return the same result as the length shown in VDIRECTORY. If you don't specify the byte size in OPEN, it will be 7 bits, which is the normal byte size for text files.

Random-access is also quite simple. Tops-20 stores files as simple character streams. So if you do (FILE-POSITION file 23), Lisp will position the file after the 23'rd byte. As with FILE-LENGTH, Lisp will use the byte size you specified when you OPENed the file. As in Common Lisp, end of line is indicated in a TOPS-20 file by characters in the text. So if your lines are different lengths there is no easy way to position to the Nth line. It is common for programs to maintain an index into the file. You can build such an index by calling FILE-POSITION when you are writing the file, to tell you where the object you are about to write will go. You can also arrange to pad short lines with extra characters, so that all lines are the same length. WARNING: Lines will be longer in the file than they are in Lisp, because end of line is one character in Lisp, but two in the file. See the next paragraph for details.

Unfortunately there is a slight discrepancy between Common Lisp and TOPS-20 conventions regarding end of line. The Common Lisp manual specifies that lines are terminated by a single end of line characters, referred to as NEWLINE. TOPS-20 normally uses a two-character sequence: carriage return (CR) followed by linefeed (LF). Thus Lisp has to turn CR/LF into NEWLINE when reading files, and NEWLINE into CR/LF when writing them. The manual allows the implementor to choose the character code for NEWLINE, but it recommends octal 12, which is LF. We have followed that recommendation. Any possible choice has its consequences. The consequences of this one is that a Lisp program will not be able to tell the difference between CRLF and a bare LF in a file. Both will show up as a single NEWLINE character. If you really have to be able to tell what your end of line is, you should read the file with READ-BYTE. This treats CR and LF just like any other character.

3.7.3. Device handling

Lisp has three different sets of I/O routines for handling external files. (There are also routines for reading from and writing to strings and the EMACS buffer.) When you OPEN a file, Lisp will choose the set of routines to use based on the the of device involved.

3.7.3.1. Disk files

If the file is on disk, Lisp will normally use a set of I/O routines that use the PMAP JSYS. These routines are capable of random access, using FILE-POSITION. They will do I/O using any byte size that you specify in the OPEN. In a few cases PMAP is not possible. If you to append to a file for which you have append-only access, of if you write to a file for which you have write-only access, the PMAP JSYS is not allowed. In this case. another set of routines is used. They use BIN and BOUT for each character individually.

3.7.3.2. Terminals

If OPEN is done to a terminal, there are several possibilities. Normally, input is done with the TEXTI JSYS and output with BOUT. TEXTI implements the normal TOPS-20 terminal handling conventions, including special actions for rubout, ^R, ^U, and ^W. In order to allow this editing. it keeps characters in a buffer until you type and end of line character (normally carriage return, but line feed, ^Z, ^L, and escape also activiate it). The Lisp program starts reading from the buffer once you have typed the end of line. At that point you can no longer make changes on that line. If you print a prompt, Lisp will automatically put it into the ^R buffer for the next read. That is, you can do something like

```
(PRINC "LISP›") (READ)
```

What you will see on the terminal is a prompt

```
LISP›
```

with the cursor waiting for input on the same line. If you type ^R or ^U, the LISP› will remain at the beginning of the line. Lisp will keep putting input and output into the ^R buffer as long as you remain on the same line. This is done on a stream by stream basis. If you open a second stream on the same terminal, you should not print a prompt from one stream and read the results from the other stream. (Such a sequence would work, however ^R would not show the prompt in the right way.)

Because output is done using BOUT for each character. Thus output

will show up on your terminal as soon as you generate it. You do not need to do anything special to force buffers to be written.

If you OPEN a terminal with a byte size of 8 (by specifying an ELEMENT-TYPE of SIGNED-BYTE or UNSIGNED-BYTE), this has a special meaning to both the operating system and Lisp. A byte size of 8 implies "binary mode". In this mode there is no echoing, and normal character processing (e.g. rubout and ^U) is not done. In some circumstances it is even possible to read ^C in binary mode. Lisp handles terminals opened this way by using simple BIN and BOUT jsyses for each character.

The choice between normal and binary mode is made when you open the file, on the basis of whether or not you specify a byte size of 8. You cannot change between these modes once the file is opened. However if you open a terminal normally, you can use the function SET-TERMINAL-MODES to change some of its parameters. These include the equivalent of the EXEC commands TERMINAL WIDTH, TERMINAL PAUSE END-OF-PAGE, TERMINAL ECHO, and RECEIVE/REFUSE SYSTEM MESSAGES. In addition, you can enable or disable PASS-ALL, TRANSLATE, and ESCAPE modes, which have no equivalent in the EXEC.

- PASS-ALL mode is very similar to the effect of opening a terminal for 8-bit I/O. It allows your program to read and write any character. ^C and other interrupt characters become normal data characters. ECHO is still done, unless you have disabled it with SET-TERMINAL-MODES. In many cases, PASS-ALL mode is not really required. If all you want is to be able to output escapes and other control characters, disabling TRANSLATE is often enough.

- TRANSLATE mode causes control characters to echo as ^ followed by a letter, and escape as $. If you disable it, then your program can output any character.

- ESCAPE mode is a designed to allow you to read the escape sequences produced by terminals with special function keys. When it is turned on, Lisp handles the escape key specially. When it sees an escape, it expects one of these special escape sequences. It does not echo the escape, nor the characters that make up the escape sequence. When it reaches the end of the escape sequence, it activates your program, as it would have if you had typed an end of line character. At the moment ESCAPE mode has no effect if you are already in PASS-ALL mode.

3.7.3.3. Other devices

If you OPEN something that is neither a disk nor a terminal, Lisp will use the BIN and BOUT monitor calls. It will do a separate call for each character you read or write. These are TOPS-20's general-purpose device-independent I/O calls, so the results should be satisfactory

for most devices. However there is no special handling for tape, networks, or other devices.

4. Reference Manual - Additional Functions and Features

This section contains documentation for all functions and options that are not part of standard Spice Lisp.

(DDT) --› NIL
Go into DDT. To exit, type RET$X. (See also section 3.6.)

(ED thing)

(EDIT thing)
See section 3.5 for documentation on the editor. EDIT is an additional function. It is just like ED, except that it does not evaluate its argument.

FEATURES - variable
The following "features" are true in this implementation. Thus you can use any of them in a #+ test: COMMON DECSYSTEM-20 TOPS-20.

GC-TRIGGER - variable
A variable, initialized to 1.0. This controls the how often a garbage collection will happen. At the end of each GC, all used space is compact. A certain amount of space above this compact, used space is then allocated for the system to grow in until the next GC. This is called "free space". Free space is computed as the number of words used * GCTRIGGER. GCTRIGGER should normally be a floating point number between 0 and 2. The default is 1.0. You will always get at least 64000 words of free space, even if the calculation just documented leads to a smaller number.

(GET-TERMINAL-MODES stream) --› mode list
Returns a list of terminal parameters, of the following form: (:BROADCAST T :ECHO T :ESCAPE NIL :PASS-ALL NIL :PAUSE T :TRANSLATE T :WRAP 80) See SET-TERMINAL-MODES for the meaning of these parameters. STREAM must be a stream that has been opened on a terminal for character I/O (i.e. not :ELEMENT-TYPE '(UNSIGNED-BYTE 8)).

(KILL-EDITOR)
Kills the subfork that has EMACS in it. You may find this necessary if EMACS because unusable for one reason or another.

(LOAD filename :PACKAGE package)
LOAD takes an additional keyword, :PACKAGE. This specifies the package into which the file will be loaded. If the file contains package specifications of its own, they will take precedence. This keyword simply rebinds *PACKAGE* for the duration of the LOAD.

#\NEWLINE - a character

See section 3.7.2 for a description of the newline convention that this implementation follows.

(OPEN file)
See section 3.7.1 for documentation on the effects of the various OPEN options. Various other I/O details are discussed in the sections following that one.

PRINT-GC-INFO - variable
A variable, initialized to NIL. It you set it to non-NIL, the garbage collector will print a message showing the total amount of free space used before and after the garbage collection. The difference between these quantities is the amount of garbage that was removed.

PROMPT - variable
A variable, initialized to "CL>". The default top level uses this is its prompt.

(SAVE filename &OPTIONAL greeting-message) --> NIL
Saves your entire core image on the file specified. The filename should probably end in .EXE. This function is similar to the SAVE command in the EXEC. However you should use this function instead of the EXEC's command, since the EXEC's command will not save the registers. Note that you need lots of disk space to use SAVE. The base core image (with just Common Lisp) is currently over 500 pages. This will go up as we load more code.

If you specify a greeting-message, it will be PRINC'ed when the core image is started.

(SET-TERMINAL-MODES stream &key parameters) --> NIL
This function allows you to control the way Lisp will handle the terminal. STREAM must be a stream that has been opened on a terminal for character I/O (i.e. not :ELEMENT-TYPE '(UNSIGNED-BYTE 8)). In many cases these setting will affect all processes using the particular terminal, not just the particular stream that is set. Here are the possible parameters. Unless otherwise stated, the default is taken from the way your terminal is set up when you enter Lisp.

:BROADCAST
non-NIL if you want your terminal to receive messages such as [You have mail from ...], and SEND's from other users. NIL to suppress these messages. Note that a privileged user can override this setting. Changing this affects all users of this terminal.

:ECHO
non-NIL for input that you type to be "echoed", assuming that you are on a full-duplex terminal. (On half-duplex terminals, the system never echos input.) NIL turns off this echo. Changing this

seems to affect other streams open on the terminal within Lisp, but not other processes than use it, except in PASS-ALL mode, where it affects only that one stream. Default is T.

:ESCAPE
non-NIL if you want escape sequences sent by ANSI terminals to be treated as terminators. Within this, it is moderately hard to read these sequences. The problem is that Lisp does not normally process input until you type carriage-return, line-feed, escape, form-feed, or ^Z. However typically you want an escape sequence to be processed immediately. This mode causes input to be processed as soon as a complete escape sequence is seen. It also turns off echoing during processing of the escape sequence. The escape sequences recognized are a superset of those accepted by the ESCAPE option in VMS. This includes all legal ANSI escape and control sequences, plus most of the sequences sent by the older VT52-compatible terminals. This affects only the one stream for which you issue it. Default is NIL. Escape processing currently does not work for pass-all mode.

:PASS-ALL
non-NIL if you want to be able to treat most special characters as ordinary data. With this turned on, rubout, ^U, etc.. are just ordinary characaters for input. Also, output characters are sent as is. That is, escape is not turned into dollar sign, control-X into ^X, etc. Interrupt characters, such as ^C. will be treated as normal data characters. This affects only the one stream for which you issue it, except that interrupt characters are turned on and off globally. Default is NIL. If echoing is turned on, you had better have opened the stream :DIRECTION :IO, since Lisp will have do the echoing explicitly.

:PAUSE
non-NIL if you want the system to wait for ^Q each time it fills your screen. This is equivalent to TERM PAUSE END-OF-PAGE in the EXEC. Changing this affects all users of this terminal.

:TRANSLATE
non-NIL if you want control characters and escape to be translated on output. That is, a control character appears as ^ followed by a letter, and escape appears as $. NIL if you want these characters to be sent as themselves. With :TRANSLATE NIL, the setting of TERM TAB or TERM NO

TAB is still obeyed. That is, if your terminal is shown as having no tabs, tabs are turned into spaces. The default is :TRANSLATE T.

:WRAP
non-NIL if you want the system to supply a carriage-return line-feed when it thinks it has reached the right margin of your terminal. This is equivalent to TERM WIDTH x in the EXEC. Turning the feature off (NIL) is equivalent to TERM WIDTH 0. It is somewhat unfortunate that there is no way to turn this off without losing the terminal width parameter. If you know the terminal width, you can specify it as the argument to turn wrapping back on. For example, you can say (SET-TERMINAL-MODES TERM :WRAP 80). Lisp will remember the terminal width that was present when you opened the terminal, if it was non-zero. (If it was zero, Lisp uses a width of 80.) If you specify an argument of T, Lisp will use this remembered value.

others
This function specifically ignores keywords that it does not know about, because other implementations of Lisp may have other keywords that do not make sense on a DECSYSTEM-20.

It is sometimes convenient to save an old terminal state, as returned by GET-TERMINAL-MODES. and then reset it. To make this easier, SET-TERMINAL-MODES may also be used in the form:

(SET-TERMINAL-MODES term modes)

In this case, modes is a list of keyword-value pairs, as returned by GET-TERMINAL-MODES.

(STEP form)
For documentation on the STEP facility, see section 3.4.

(%TOP-LEVEL) - never returns
For documentation on customizing the top level, see section 3.6.

(TRACE function)
For documentation on the TRACE facility, see section 3.3.

5. Differences between Spice Lisp and Common Lisp

The section is really intended for the benefit of implementors and maintainers. It describes the general nature of the changes we have had to make to the Spice Lisp system code in order to use it as part of Common Lisp. Such changes should not be necessary for user code, so this should not affect normal users.

Unfortunately, we have not been able to use very many of the Spice Lisp files unmodified. However in many cases the changes take only 5 minutes or so to put in. In general, our data representations are quite similar. %SP-TYPE converts the internal data type code to the correct Spice Lisp type number. Thus there are few changes necessary due to differences in types. Most of the changes are due to the fact that we implement more in the kernel than Spice Lisp implements in microcode. Here are the major things to look for in converting a file:

- Look for (PRIMITIVE and %SP-. Spice Lisp is in the process of changing its primitives. Some of them are %SP-foo and others are (PRIMITIVE foo). We have normally used the old %SP names, although we have HEADER-REF and HEADER-LENGTH without the %SP. We do not have PRIMITIVE at all. When Spice Lisp changes completely to using the (PRIMITIVE format, we should define PRIMITIVE as a macro. This would eliminate a lot of the conversion.

- Look for code of the form (DEFUN CAR (X) (CAR X)). This is used to provide Lisp definitions for the Spice Lisp primitives. Since our kernel uses normal Lisp calling conventions, such definitions are not needed, and should be deleted.

- Look for functions that we define in the kernel. Large parts of HASH, EVAL, PRINT, READ, and FILESYS are implemented in the kernel. We have tried to be consistent with Spice Lisp in the function names that we used for kernel code. So often you just have to remove those functions that are already in the kernel. In some cases it was inconvenient to do all of the argument processing in the kernel. so I supply a small Lisp function to do that. For example, most of OPEN is in the kernel. But the kernel function is called %SP-OPEN. In FILESYS.CLISP the actual OPEN is defined in Lisp. It simply does some defaulting and then calls %SP-OPEN. Arithmetic is done almost entirely in the kernel.

- Some of the functions, particularly in FILESYS and MISC, are inherently system-dependent.

- Some of our changes are simply bug fixes.