CARNEGIE-MELLON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

SPICE PROJECT

Spice Lisp User's Guide

Edited by Scott E. Fahlman and Monica J. Cellio

9 November 1983

Companion to the Excelsior Edition of the Common Lisp Manual

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-78-C-1551. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the U.S. Government.

Chapter 1

Introduction

Common Lisp is a new dialect of Lisp, closely related to Maclisp and Lisp Machine Lisp. Common Lisp was developed in response to the need for a modern, stable, well documented dialect of Lisp that can be implemented efficiently on a variety of machine architectures.

Spice Lisp is the implementation of Common Lisp for microcodable personal machines running CMU's Spice computing environment. At present, Spice runs only on the Three Rivers Computer Corporation's PERQ; implementations for other machines are planned but not yet under way. Compatible versions of Common Lisp will soon be available for the DEC Vax, under both VMS and Unix, the Decsystem-20 with extended addressing, and the Symbolics 3600.

The central document for users of any Common Lisp implementation is the *Common Lisp Reference Manual*, by Guy L. Steele Jr. All implementations of Common Lisp must conform to this standard. However, a number of design choices are left up to the implementor, and implementations are free to add to the basic Common Lisp facilites. This document covers those choices and features that are specific to the Spice Lisp implementation. The *Common Lisp Reference Manual* and *Spice Lisp User's Guide*, taken together, should provide everything that the user of Spice Lisp needs to know.

For now, a number of documents describing useful library modules that run in Spice Lisp are included here. Once there are enough of these, the documents will be moved into a separate document on the Spice Lisp Program Library.

Spice Lisp is currently undergoing intensive tuning and development. For the next year or so, at least, new releases will be appearing frequently. This document will be modified for each major release, so that it is always up to date. Users of Spice Lisp at CMU should watch the SPICE and CLISP bulleting boards for release announcements, pointers to updated documentation files, and other information of interest to the user community.

1.1. Obtaining and Running Spice Lisp

In order to run Spice Lisp, you must have a Perq 1a or Perq 2 with 16K control store. You must also have an up-to-date Accent system. Use the update program to get the current release of Accent. Then, decide where you want the Spice Lisp files to live. There must be at least 3500 pages free in the partition you wish to put Spice Lisp in. It is suggested that you make a subdirectory called **slisp** in the **user** partition. Path to the directory you want to put Spice Lisp in, then run the update program on the directory /usr/spice/slisp/**run**. When Spice Lisp is on your Perq, put the directory that it resides in on your search list and then just type **lisp** to the Accent shell.

Chapter 2

Implementation Dependent Design Choices

Several design choices in Common Lisp are left to the individual implementation. This chapter contains a partial list of these topics and the choices that are implemented in Spice Lisp.

2.1. Numbers

Currently, short-floats and single-floats are the same, and long-floats and double-floats are the same. Short floats use an immediate (non-consing) representation with 8 bits of exponent and a 21-bit mantissa. Long floats are 64-bit consed objects, with 12 bits of exponent and 53 bits of mantissa. All of these figures include the sign bit and, for the manitssa, the "hidden bit". The long-float representaiton conforms to the 64-bit IEEE standard, except that we do not support all the exceptions, negative 0, inifinities, and the like.

Fixnums are stored as 28-bit two's complement integers, including the sign bit. The most positive fixnum is 2^{27} - 1, and the most negative fixnum is -2^{27}. An integer outside of this range is a bignum.

2.2. Characters

Spice Lisp characters have 8 bits of `code`, 8 bits of `font`, and 8 `control` bits. The font bits are not used, and only 4 of the control bits are used: (`control`, `meta`, `super`, and `hyper`).

The control bit functions `Control`, `Meta`, `Super`, and `Hyper` are defined as in the *Common Lisp Manual*. The Perq keyboard does not produce these and Accent does not pass them to Spice Lisp, but programs can use these internally.

2.3. Vector Initialization

If no `:initial-value` is specified, vectors of Lisp objects are initialized to `nil`, and vectors of integers are initialized to 0.

2.4. Packages

Common Lisp requires four built-in packages: `lisp`, `user`, `keyword`, and `system`. In addition to these, Spice Lisp has separate packages for `hemlock` (the editor) and `compiler`.

2.5. The Editor

The `ed` function will invoke the Hemlock Editor.

2.6. Time Functions

There are at present no time functions in Spice Lisp, due to the difficulty of getting such information from Accent. This is being worked on.

2.7. System Dependent Constants

The following constants are defined in Spice Lisp.

boole-clr	0
boole-set	1
boole-1	2
boole-2	3
boole-c1	4
boole-c2	5
boole-and	6
boole-ior	7
boole-xor	8
boole-eqv	9
boole-nand	10
boole-nor	11
boole-andc1	12
boole-andc2	13
boole-orc1	14

boole-orc2	15
most-positive-fxnum	134217727
most-negative-fixnum	-134217728
most-positive-short-float	1.7014e38
least-positive-short-float	0.0
least-negative-short-float	0.0
most-negative-short-float	-1.7014e38
most-positive-single-float	1.701411e38
least-positive-single-float	0.0
least-negative-single-float	0.0
most-negative-single-float	-.85071e38
most-positive-double-float	0.0d0
least-positive-double-float	0.0d0
least-negative-double-float	0.0d0
most-negative-double-float	0.0d0
most-positive-long-float	0.0d0
least-positive-long-float	0.0d0
least-negative-long-float	0.0d0
most-negative-long-float	0.0d0
short-float-epsilon	4.76837e-7
single-float-epsilon	4.76837e-7
double-float-epsilon	0.0d0
long-float-epsilon	0.0d0
short-float-negative-epsilon	4.76837e-7
single-float-negative-epsilon	4.76837e-7

double-float-negative-epsilon	0.0d0
long-float-negative-epsilon	0.0d0
char-code-limit	256
char-font-limit	256
char-bits-limit	256
char-control-bit	1
char-meta-bit	2
char-super-bit	4
char-hyper-bit	8
array-rank-limit	134217727
internal-time-units-per-second	1

Chapter 3

Debugging Tools

By Jim Large

3.1. The Break Loop

The break loop is a read-eval-print loop which is similar to the normal lisp top level. It can be called from any lisp function to allow the user to interact with the lisp system. When the user gives the command to exit the break loop, he may choose an arbitrary value for it to return.

When a lisp expression is typed in at the break loop's prompt, it is usually evaluated and printed. However, there are three special expressions which are recognized as break loop commands, and which are not evaluated.

`$G` Typing this symbol causes a throw to the lisp top level: The current computation is aborted, and all bindings are unwound.

`$P` Typing this symbol causes the break loop to return `nil`.

`(RETURN` *form*`)` Typing this expression causes the break loop to evaluate *form* and return the result(s).

The dollar sign character in the symbols `$P` and `$G` is intended to be the (escape) character -- ascii 27. For compatibility with the VAX VMS operating system, real dollar signs will be recognized also.

When the break loop is called, it tries to make sure that terminal interaction will be possible. All of the standard input output streams, `*standard-input*`, `*standard-output*`, `*error-output*`, `*query-io*`, and `*trace-output*` are bound to `*terminal-io*` for the duration of the break loop; and the state of the single stepper is bound to "off".

`break` *tag* `&optional` *condition* [*Macro*]

The `break` macro returns a form which prints the message "Breakpoint *tag*" to `*terminal-io*` and then invokes the break loop. If *condition* is present, then the form evaluates it and tests the result. If the result is `nil`, then the form returns `nil`; otherwise, the form prints the tag and invokes the break loop. *tag* is never evaluated.

3.1.1. Cleaning Up

The break loop is called by the system error handlers. Since errors can happen unexpectedly, the break loop provides a mechanism for cleaning up any unusual state that a program may have caused.

`*error-cleanup-forms*` *[Variable]*

A list of lisp forms which will be evaluated for side effect when a break loop is invoked. Whenever a break loop is entered, `*error-cleanup-forms*` will be bound to `nil`, and then the forms which were its previous value will be `eval`ed for side effect. There is no way to have the side effects undone when the break loop returns, and if any of the cleanup forms causes an error, the result can not be guaranteed.

As an example, a program that puts the terminal in an unusual mode might want to do something like this.

```
(let ((*error-cleanup-forms*
        (cons '(progn <code to restore terminal>)
              *error-cleanup-forms*)))
  <code to mess up terminal>
  .
  .
  .)
```

3.2. Function Tracing

The tracer causes selected functions to print their arguments and their results whenever they are called. Options allow conditional printing of the trace information and conditional breakpoints on function entry.

`trace &rest` *specs* *[Macro]*

Invokes tracing on the specified functions,[1] and pushes their names onto the global list in `*traced-function-list*`. Each *spec* is either the name of a function, or the form

```
(function-name
    trace-option-name value
    trace-option-name value
    ...)
```

If no *specs* are given, then `trace` will return the list of all currently traced functions, `*traced-function-list*`.

If a function is traced with no options, then each time it is called, a single line containing the name of the function, the arguments to the call, and the depth of the call will be printed on the stream `*trace-output*`. After it returns, another line will be printed which contains the depth of the call and all of the return values. The lines are indented to highlight the depth of the calls.

Trace options can cause the normal printout to be suppressed, or cause extra information to be printed. Each traced function carries its own set of options which is independent of the options

---

[1] `Trace` does not work on macros or special forms yet.

given for any other function. Every time a function is specified in a call to trace, all of the old options are discarded. The available options are:

`:condition` A form to eval before before each call to the function. Trace printout will be suppressed whenever the form returns `nil`.

`:break` A form to eval before each call to the function. If the form returns non `nil`, then a breakpoint loop will be entered immediately before the function call.

`:break-after` Like `:break`, but the form is evaled and the break loop invoked after the function call.

`:break-all` A form which should be used as both the `:break` and the `:break-after` args.

`:wherein` A function name or a list of function names. Trace printout for the traced function will only occur when it is called from within a call to one of the `:wherein` functions.

`:print` A list of forms which will be evaluated and printed whenever the function is called. The values are printed one per line, and indented to match the other trace output. This printout will be suppressed whenever the normal trace printout is suppressed.

`:print-after` Like `:print` except that the values of the forms are printed whenever the function exits.

`:print-all` The arg is used as the arg to both `:print` and `:print-after`.

`untrace &rest` *function-names* [*Macro*]

Turns off tracing for the specified functions, and removes their names from `*traced-function-list*`. If no *function-names* are given, then all functions named in `*traced-function-list*` will be untraced.

`*traced-function-list*` [*Variable*]

A list of function names which is maintained and used by `trace`, `untrace`, and `untrace-all`. This list should contain the names of all functions which are currently being traced.

`*trace-prinlevel*` [*Variable*]

`*trace-prinlength*` [*Variable*]

`*Prinlevel*` and `*prinlength*` are bound to `*trace-prinlevel*` and `*trace-prinlength*` when printing trace output. The forms printed by the `:print` options are affected also. `*Trace-prinlevel*` and `*trace-prinlength*` are initially set to `nil`.

`*max-trace-indentation*` [*Variable*]

The maximum number of spaces which should be used to indent trace printout. This variable is initially set to some reasonable value.

3.2.1. Encapsulation Functions

The encapsulation functions provide a clean mechanism for intercepting the arguments and results of a function.[2] `Encapsulate` changes the function definition of a symbol, and saves it so that it can be restored later. The new definition normally calls the original definition.

The original definition of the symbol can be restored at any time by the `unencapsulate` function. `Encapsulate` and `unencapsulate` allow a symbol to be multiply encapsulated in such a way that different encapsulations can be completely transparent to each other.

Each encapsulation has a type which may be an arbitrary lisp object. If a symbol has several encapsulations of different types, then any one of them can be removed without affecting more recent ones. A symbol may have more than one encapsulation of the same type, but only the most recent one can be undone.

`encapsulate` *symbol type body* [*Function*]

Saves the current definition of *symbol*, and replaces it with a function which returns the result of evaluating the form, *body*. *Type* is an arbitrary lisp object which is the type of encapsulation.

When the new function is called, the following variables will be bound for the evaluation of *body*:

`argument-list`
: A list of the arguments to the function.

`basic-definition`
: The unencapsulated definition of the function.

The unencapsulated definition may be called with the original arguments by including the form

```
(apply basic-definition argument-list)
```

`Encapsulate` always returns *symbol*.

`unencapsulate` *symbol type* [*Function*]

Undoes *symbol*'s most recent encapsulation of type *type*. *Type* is compared with `eq`. Encapsulations of other types are left in place.

`encapsulated-p` *symbol type* [*Function*]

Returns `t` if *symbol* has an encapsulation of type *type*. Returns `nil` otherwise. *type* is compared with `eq`.

---

[2] Encapsulation does not work for macros or special forms yet.

3.3. Single Stepper

`step` *form* [*Function*]

Evaluates *form* with single stepping enabled, or if *form* is `t`, enables stepping until explicitly disabled. Stepping can be disabled by quitting to the lisp top level, or by evaluating the form `(step ())`.

While stepping is enabled, every call to `eval` will prompt the user for a single character command. The prompt is the form which is about to be `eval`ed. It is printed with `*prinlevel*` and `*prinlength*` bound to `*step-prinlevel*` and `*step-prinlength*`. All interaction is done through the stream `*query-io*`.

The commands are:

n (next)	Evaluate the expression with stepping still enabled.
s (skip)	Evaluate the expression with stepping disabled.
q (quit)	Evaluate the expression, but disable all further stepping inside the current call to `step`.
p (print)	Print current form. (does not use `*step-prinlevel*` or `*step-prinlength*`.)
b (break)	Enter break loop, and then prompt for the command again when the break loop returns.
e (eval)	Prompt for and evaluate an arbitrary expression. The expression is evaluated with stepping disabled.
? (help)	Prints a brief list of the commands.
r (return)	Prompt for an arbitrary value to return as result of current call to eval.
g	Throw to top level.

`*step-prinlevel*` [*Variable*]
`*step-prinlength*` [*Variable*]

`*Prinlevel*` and `*Prinlength*` are bound to these values when the current form is printed. `*Step-prinlevel*` and `*step-prinlength*` are initially bound to some small value.

`*max-step-indentation*` [*Variable*]

Step indents the prompts to highlight the nesting of the evaluation. This variable contains the maximum number of spaces to use for indenting. It is initially set to some reasonable number.

3.4. The Debugger

The debugger is an interactive command loop which allows a user to examine the active call frames on the Lisp function call stack. If it is invoked from an error breakpoint, it can show the function calls which led up to the error.

Only one stack frame, the current frame, can be inspected at any given time. The command loop prints the frame number of the current frame as a prompt, and then reads a lisp expression from the terminal. `debug` tries to use the expression as a command, but if it fails, then it evals and prints the expression like a breakpoint loop. Terminal input and output are done by binding `*standard-input*` and `*standard-output*` to `*terminal-io*`.

3.4.1. Movement Commands

These commands move to a new stack frame, and print out the name of the function and the values of its arguments in the style of a lisp function call. `*debug-prinlevel*` and `*debug-prinlength*` affect the style of the printing.

Up is toward the most recent function invocation, and lower frame numbers. Down is toward older function calls and higher frame numbers.

Visible frames are those which have not been hidden by the **hide** command which is described below. The special variable `*debug-ignored-functions*` contains a list of function names which are hidden by default.

The commands are:

u Move up to the next higher visible frame.

d Move down to the next lower visible frame.

t Move to the highest visible frame.

b Move to the lowest visible frame.

(frame *n*) Move to frame number *n* whether it is visible or not.

3.4.2. Inspection Commands

These commands print information about the current frame and the current function.

? `describe`'s the current function.

a Lists the arguments to the current function. The values of the arguments are printed along with the argument names.

l Lists the local variables in the current function. The values of the locals are printed, but their names are no longer available.

p Redisplays the current function call as it would be displayed by moving to this frame.

pp Redisplays the current function call using `*prinlevel*` and `*prinlength*` instead of `*debug-prinlevel*` and `*debug-prinlength*`.

(value *symbol*) Prints the value of *symbol* in the current binding context. *symbol* is either a special variable, or the name of an argument to the current function.

(local *n*) Prints the value of the *n*th local variable in the current frame.

3.4.3. Other Commands

h Prints a brief list of commands on the terminal.

q Causes `debug` to return `nil`.

(return &rest *values*)
Forces the current function to return zero or more values. If the function was not called for multiple values, then attempts to return other than one value will be prevented.

(backtrace *options*)
Prints a history of function calls. The printing is controlled by `*debug-prinlevel*` and `*debug-prinlength*`. Only those frames which are considered visible by the frame movement commands will be shown. Currently, there are no *options*.

(hide *options*) Makes the stack frames described by *options* be invisible to the frame movement commands. The first *option* is a subcommand which may be one of:

`package` Followed by one or more package names. Calls to functions in the named packages will not be visible.

`function` Followed by one or more function names. Calls to the named functions will not be visible.

`compiled` Calls to compiled functions will not be visible.

`interpreted` Calls to interpreted functions will not be visible.

`lambdas` Calls to lambda expressions will not be visible.

(show *options*) Options are the same as for the **hide** command. **Show** is the inverse operation. **Showing** a particular set of functions does not necessarily make them all visible; some of them may still be hidden for other reasons.

`debug` *[Function]*

Invokes the debugger. Always returns `nil`.

`*debug-prinlevel*` [*Variable*]
`*debug-prinlength*` [*Variable*]

`*Prinlevel*` and `*prinlength*` are bound to these values during the execution of some debug commands. When evaluating arbitrary expressions in the debugger, the normal `*Prinlevel*` and `*prinlength*` are in effect. These variables are initially set to some small number.

`*debug-ignored-functions*` [*Variable*]

A list of functions which are hidden by default. These functions can be made visible with the `debug` command **show**.

3.5. Random Features

`describe` *object* [*Function*]

The `describe` function prints useful information about *object* on `*standard-output*`. For any object, `describe` will print out the type. Then it prints other information based on the type of `object`. The types which are presently handled are:

`function` `describe` prints a list of the function's name (if any) and its formal parameters. If the function has documentation, then the documentation string will be printed.

`symbol` The symbol's value, properties, and documentation are all printed. If the symbol has a function definition, then the function is described.

Chapter 4

The Compiler

4.1. Calling the Compiler

Functions are compiled using either the `compile` or `compile-file` functions. Both operate as documented in the *Common Lisp Reference Manual.* The `compile-file` function takes the following keyword arguments: `:output-file`, `:cold-file`, `:lap-file`, and `:error-file`. These take either the name of a file, as a string, or the symbol `t`, which tells the system to make up an appropriate name by replacing the type field of the input file name. If any of these arguments is NIL, no output of that type is created. By default, only the output file and error file are created.

4.2. Open and Closed Coding

When a function call is "open coded," inline code whose effect is equivalent to the function call is substituted for that function call. When a function call is "closed coded", it is usually left as is, although it might be turned into a call to a different function with different arguments. As an example, if `nthcdr` were to be "open coded" then

```
(nthcdr 4 foobar)
```

might turn into

```
(cdr (cdr (cdr (cdr foobar))))
```

or even

```
(do ((i 0 (1+ i))
             (list foobar (cdr foobar)))
            ((= i 4) list)).
```

If `nth` is "closed coded"

```
(nth x 1)
```

might stay the same, or turn into something like:

```
(car (nthcdr x 1)).
```

4.3. Compiler Switches

Several compiler switches are available which are not documented in the *Common Lisp Manual*. Each is a global special. These are described below.

`*peep-enable*` If this switch is non-nil, the compiler runs the peephole optimizer. The optimizer makes the compiled code faster, but the compilation itself is slower. `*peep-enable*` defaults to `t`.

`*peep-statistics*`
: If this switch is non-nil, the effectiveness of the peephole optimizer (number of bytes before and after optimization) will be reported as each function is compiled. `*peep-statistics*` defaults to `t`.

`*inline-enable*`
: If this switch is non-nil, then functions which are declared to be inline are expanded inline. It is sometimes useful to turn this switch off when debugging. `*inline-enable*` defaults to `t`.

`*open-code-sequence-functions*`
: If this switch is non-nil, the compiler tries to translate calls to sequence functions into do loops, which are more efficient. It defaults to `t`.

`*nthcdr-open-code-limit*`
: This is the maximum size an `nthcdr` can be to be open coded. In other words, if `nthcdr` is called with `n` equal to some constant less than or equal to the `*nthcdr-open-code-limit*`, it will be open coded as a series of nested `cdr`'s. `*nthcdr-open-code-limit*` defaults to 10.

`*complain-about-inefficiency*`
: If this switch is non-nil, the compiler will print a message when certain things must be done in an inefficient manner because of lack of declarations or other problems of which the user might be unaware. This defaults to `nil`.

`*eliminate-tail-recursion*`
: If this switch is non-nil, the compiler attempts to turn tail recursive calls (from a function to itself) into recursion. This defaults to `t`.

`*all-rest-args-are-lists*`
: If non-nil, this has the effect of declaring every `&rest` arg to be of type list. (They all start that way, but the user could alter them.) It defaults to `nil`.

`*verbose*` If this switch is `nil`, only true error messages and warnings go to the error stream. If non-nil, the compiler prints a message as each function is compiled. It defaults to `t`.

`*check-keywords-at-runtime*`
: If non-nil, compiled code with `&key` arguments will check at runtime for unknown keywords. This is usually left on and defaults to `t`.

4.4. Declare switches

Not all switches for `declare` are processed by the compiler. The `ftype` and `function` declarations are currently ignored.

The `optimize` declaration controls some of the above switches:

- `*peep-enable*` is on unless `cspeed` is greater than `speed` and `space`.
- `*inline-enable*` is on unless `space` is greater than `speed`.
- `*open-code-sequence-functions*` is on unless `space` is greater than `speed`.
- `*eliminate-tail-recursion*` is on if speed is greater than space.

Chapter 5
Efficiency

By Rob Maclaclan

In Spice Lisp on the Perq, as is any language on any computer, the way to get efficient code is to use good algorithms and sensible programming techniques, but to get the last bit of speed it is helpful to know some things about the language and its implementation. This chapter is a summary of various hidden costs in the implementation and ways to get around them.

5.1. Compile Your Code

In Spice Lisp, compiled code typically runs at least sixty times faster than interpreted code. Another benefit of compiling is that it catches many typos and other minor programming errors. Many lisp programmers find that the best way to debug a program is to compile the program to catch simple errors, then debug the code *interpreted*, only actually using the compiled code once the program is debugged.

Another benefit of compilation is that compiled (*sfasl*) files load significantly faster, so it is worthwhile compiling files which are loaded many times even if the speed of the functions in the file is unimportant.

Do Not be concerned about the performance of your program until you see its speed compiled -- some techniques that make compiled code run faster make interpreted code run slower.

5.2. Avoid Unnecessary Consing

Consing is the Lispy name for allocation of storage, as done by the `cons` function, hence its name. `cons` is by no means the only function which conses -- so does make-array and many other functions. Even worse, the Lisp system may decide to cons furiously when you do some apparently innocent thing.

Consing hurts performance in the following ways:

- Consing reduces your program's memory access locality, increasing paging activity.

- Consing takes time just like anything else.

- Any space allocated eventually needs to be reclaimed, either by garbage collection or killing your

Lisp.

Of course you have to cons sometimes, and the Lisp implementors have gone to considerable trouble to make consing and the subsequent garbage collection as efficient as possible. In some cases strategic consing can improve speed. It would certainly save time to allocate a vector to store intermediate results which are used hundreds of times.

5.3. Do, Don't Map

One of the programming styles encouraged by Lisp is a highly applicative one, involving the use of mapping functions and many lists to store intermediate results. To compute the sum of the square-roots of a list of numbers, one might say:

```
(apply #'+ (mapcar #'sqrt list-of-numbers))
```

This programming style is clear and elegant, but unfortunately results in slow code. There are two reasons why:

- The creation of lists of intermediate results causes much consing (see 5.2).

- Each level of application requires another scan down the list. Thus, disregarding other effects, the above code would probably take twice as long as a straightforward iterative version.

An example of an iterative version of the same code:

```
(do ((num list-of-numbers (cdr num))
     (sum 0 (+ (sqrt (car num)) sum)))
    ((null num) sum))
```

Once you feel in you heart of hearts that iterative Lisp is beautiful then you can join the ranks of the Lisp efficiency fiends.

5.4. Think Before You Use a List

Although Lisp's creator seemed to think that it was for LISt Processing, the astute observer may have noticed that the chapter on list manipulation makes up less that ten percent of the COMMON LISP manual. The language has grown since Lisp 1.5, and now has other data structures which may be better suited to tasks where lists might have been used before.

5.4.1. Use Vectors

Use Vectors and use them often. Lists are often used to represent sequences, but for this purpose vectors have the following advantages:

- A vector takes up less space than a list holding the same number of elements. The advantage may vary from a factor of two for a general vector to a factor of sixty-four for a bit-vector. Less space means less consing (see 5.2).

- Vectors allow constant time random-access. You can get any element out of a vector as fast as you can get the first out of a list if you make the right declarations.

The only advantage that lists have over vectors for representing sequences is that it is easy to change the length of a list, add to it and remove items from it. Likely signs of archaic, slow lisp code are `nth` and `nthcdr` -- if you are using these function you should probably be using a vector.

5.4.2. Use Structures

Another thing that lists have been used for is the representation of record structures. Often the structure of the list is never explicitly stated and accessing macros are not used, resulting in impenetrable code such as:

```
(rplaca (caddr (cadddr x)) (caddr y))
```

The use of `defstruct` structures can result in much clearer code, one might write instead:

```
(setf (beverage-flavor (astronaut-beverage x)) (beverage-flavor y))
```

Great! But what does this have to do with efficiency? Since structures are based on vectors, the `defstruct` version would likewise take up less space and be faster to access. Don't be tempted to try and gain speed by trying to use vectors directly, since the compiler knows how to compile faster accesses to structures than you could easily do yourself. Note that the structure definition should be compiled before any uses of accessors so that the compiler will know about them.

5.4.3. Use Hashtables

In many applications where association lists (alists) have been used in the past, hashtables would work much better. An alist may be preferable in cases where the user wishes to rebind the alist and add new values to the front, shadowing older associations. In most other cases, if an alist contains more than a few elements, a hashtable will probably do the job faster. If the keys in the hashtable are objects that can be compared with `eql` or better yet `eq`, then hashtable access will be speeded up by specifying the correct function as the `:test` argument to `make-hashtable`.

5.4.4. Use Bit-Vectors

Another thing that lists have been used for is set manipulation. In some applications where there is a known, reasonably small universe of items Bit-Vectors could be used instead. This is much less convenient than using lists, because instead of symbols, each element in the universe must be assigned a numeric index into the bit vector. If the universe is vary small -- twenty-eight items or less -- then you can represent your set as bits in a fixnum and use *logior* and so on, to get immense speed improvements.

Note: right now, boolean opeartions on bit-vectors are very slow, since one bit is processed at a time instead of 16 or 32 bits at once. This will be fixed soon. In the meantime, boolean operations on bignums are faster than those on bit-vectors.

5.5. Simple Vs Complex Arrays

Spice Lisp has two different representations for arrays, one which is accessed rapidly in microcode and one which is accessed much more slowly in Lisp code. The class of arrays which can be represented in the fast form corresponds exactly to the *one dimensional simple-arrays*, as defined in the COMMON LISP manual. Included in this group are the types `simple-string`, `simple-vector` and `simple-bit-vector`.

Declare Your Vector Variables -- If you don't the compiler will be forced to assume you are using the inefficient form of vector. Example:

```
(defun iota (n)
  (let ((res (make-vector n)))
    (declare (simple-vector n))
    (dotimes (i n)
      (setf (aref res i) i))
    res))
```

Warning: if you declare things to be simple when they are not, incorrect code will be generated and hard-to-find bugs will result. It is worthwhile to note, however, that system functions which create vectors will always create simple-arrays unless you force them to do otherwise.

5.6. To Call or Not To Call

The usual Lisp style involves small functions and many function calls; for this reason Lisp implementations strive to make function calling as inexpensive as possible. Spice Lisp is fairly successful in this respect. Function calling is not vastly more expensive than other instructions, and is certainly faster than procedure calling in Perq Pascal.

For this reason you should not be overly concerned about function-call overhead in your programs. *However*, function calling does take time, and thus is not the kind of thing you want going on in the inner loops of your program. Where removing function calling is desirable you can use the following techniques:

Write the code in-line
: This is not a very good idea, since it results in obscure code, and spreads the code for a single logical function out everywhere, making changes difficult.

Use macros
: A macro can be used to achieve the effect of a function call without the function-call overhead, but the extreme generality of the macro mechanism makes them tricky to use. If macros are used in this fashion without some care, obscure bugs can result.

Use inline functions
: This often the best way to remove function call overhead in COMMON LISP. A function may be written, and then declared inline if it is found that function call overhead is excessive. Writing functions is easier that writing macros, and it is easier to declare a function inline than to convert it to a macro. Note that the compiler must process first the inline declaration, then the definition, and finally any calls which are to be open coded for the inline expansion to take place.

Note that any of the above techniques can result in bloated code, since they have the effect of duplicating the same instructions many places. If code becomes very large, paging may increase, resulting in a significant slowdown. Inline expansion should only be used where it is needed. Note that the same function may be called normally in some places and expanded inline in other places.

5.7. Keywords and the Rest

COMMON LISP has very powerful argument passing mechanisms. Unfortunately, two of the most powerful mechanisms, rest arguments and keyword arguments, have a serious performance penalty in Spice Lisp.

The main problem with rest args is that the microcode must cons a list to hold the arguments. If a function is called many times or with many arguments, large amounts of consing may occur.

Keyword arguments are built on top of the rest arg mechanism, and so have all the above problems plus the problem that a significant amount of time is spent parsing the list of keywords and values on each function call.

Neither problem is serious unless thousands of calls are being made to the function in question, so the use of argument keywords and rest args is encouraged in user interface functions.

Another way to avoid keyword and rest-arg overhead is to use a macro instead of a function, since the rest-arg and keyword overhead happens at compile time and not necessarily at runtime. If the macro-expanded form contains no keyword or rest arguments, then it is perfectly acceptable to use keywords and rest-args in macros which appear in inner loops.

Note: the compiler open-codes most heavily-used system functions which have keyword or rest arguments, so that no run-time overhead is involved.

Optional arguments have no significant overhead.

5.8. Numbers

Spice Lisp provides five types of numbers for your enjoyment:

- fixnums bignums ratios short-floats long-floats

Only short-floats and fixnums have an immediate representation; the rest must be consed and garbage-collected later. In code where speed is important, you should use only fixnums and short-floats unless you have a real need for something else. Since `most-positive-fixnum` is more than one hundred million, you shouldn't need to use bignums unless you are counting the reasons to use Lisp instead of Pascal. Unfortunately the amount of floating point precision that will fit in twenty-eight bits is severely limited, so there are reasonable problems which require the use of long-floats.

Another feature of ratios and bignums which will keep you entertained for hours is that operations on these numbers are written in Lisp, not microcode; this results in orders of magnitude slower execution.

Printing of long-floats is painfully slow -- around three seconds. While you wait, consider that the float printing algorithm is the only known correct float printing method. Other methods run in real time, but they lose precision in the low-order digits.

5.9. Timing

Everyone knows that the first step in improving a program's performance is to make extensive timings to find which code is time-critical. Unfortunately Spice Lisp currently has no timing functions. The recommended timing method is to write a *compiled* driver function which calls the function to be tested a few hundred times. If one measures the total time with a watch or by a systat and divides by the number of iterations, then fairly accurate statistics can be collected.

Chapter 6

The Alien Data Type

By Jim Large and Dan Aronson

A problem that arises in many Common Lisp implementations is dealing with the complex structured records or messages that are exchanged at the interface between Lisp and the outside world. Such alien data structures will typically be collections of integers, floating point numbers, strings, boolean flags, bit vectors, enumerated types represented as small integers, and so on. All of these types have some rough correspondence with internal Lisp data-types, but at the time of their arrival and departure they will be in whatever implementation-dependent format is expected by the alien software, and the Lisp garbage collector must treat the alien object as an unstructured vector of bits or bytes.

Given a knowledge of the structure of an alien record, it is relatively easy for the Lisp-level code to convert each field of the message into the corresponding Lisp form, but we want this knowledge to be concentrated in one place so that changes in the external message format can be easily accommodated by the Lisp code. What is needed is a convenient form for specifying how the alien record is to be parsed and packed and how each of its fields is to be interpreted as a Lisp object. In a manner similar to `defstruct`, this specification will be processed to create a family of field-accessing and field-altering macros that perform the proper translations, in addition to doing the access. Thus, once the structure of the alien record has been specified, it is no harder to access than the fields of a `defstruct`.

This new facility is built into Vax Common Lisp and Spice Lisp.

6.1. The Alien-Structure Data Type

There is a new data-type called `alien-structure`. This is just a new structure-type defined by `defstruct`. The alien structure contains a name (a lisp symbol), a length (number of 8-bit bytes in the data vector), and a pointer to the actual blob of uninterpreted bits. In Spice Lisp and Vax Common Lisp, this blob is an packed-fixnum vector (a U-Vector, in our internal parlance) of 8-bit bytes; other implementations might have to use a bit-vector for this. One could ask `typep` if an object is an `alien-structure`, and could access the innards via `alien-structure-name` and `alien-structure-data`. One can also find the length of the data area (in bytes) using the macro `alien-structure-length`.

Alien structures are created by a macro, `def-alien-structure`, that parallels `defstruct` in form:

```
(def-alien-structure (name option1 option2 ...)
  field-description-1
  field-description-2
  ...)
```

where the field descriptions are of the form

```
(field-name alien-field-type start end
  field-option-name-1 field-option-value-1
  field-option-name-2 field-option-value-2
  ...)
```

The options in `def-alien-structure` are a subset of those allowed in `defstruct`: `:conc-name`, `:constructor`, `:predicate`, `:print-function`, and `:eval-when`. Alien structures are always named and the user cannot specify a `:type` option or any of the array options.

In addition, there is a `:length` option that takes as its argument the length of the data area in 8-bit bytes. This is used when new instances of this structure-type are created within Lisp by the constructor macro. If `:length` is not specified, the length defaults to the maximum of the *end* values of the fields making up the structure, ignoring any fields with *end* values of `nil`. When alien structures are read in from outside the Lisp, `:length` controls the allocated length of the data vector, but the actual length (as reported by `alien-structure-length`) is set by the size of the incoming block of data. An error will be signalled if the incoming data block does not fit within the allocated length. If no `:length` is specified in this case, the default is to allocate a vector the same size as the incoming data block.

Each field has a name, perhaps modified by `:conc-name`. The field options are `:invisible` and `:read-only`, as in `defstruct`, plus

`:default-value`. The latter is a value that is placed into the slot at the time the alien structure is created, if no other value is specified at that time. This value is inserted into the alien structure as if by `setf`, and it is therefore processed by the field-type conversions (see below). If no `:default-value` is specified, the field is initialized to 0. (This initialization is only done if the data block is created within the Lisp; if it arives from outside, the bits are left alone unless specifically altered by the user.)

The *start* and *end* values for a field indicate where, in the alien structure's data area, the field is to be found. These numbers are in terms of bytes. As usual, they are zero-based, *start* is inclusive and *end* is exclusive. If the *end* value is `nil`, the field has no fixed length, but runs from the specified *start* to the *end* of the data block, as indicated by `alien-structure-length`. When such a field is written into, the `alien-structure-length` will be adjusted to reflect the new *end* of the field; however, any attempt to extend the field beyond the allocated length of the data vector will signal an error.

It is possible for two fields to overlap; sometimes this will be useful when one field wants to be interpreted in two different ways. Obviously, if two overlapping fields are written into, the later write clobbers the results of the earlier one. It is also possible to have gaps between the defined fields; these would correspond to parts of an incoming message that are uninteresting to the Lisp program, for example. Such gaps are initialized to 0 when the alien structure is created within the Lisp, unless the block of data comes in from outside. If the block comes from the outside, the bits in the inter-field gaps are not altered. Fields may appear in any order within `def-alien-structure`.

In the rare cases where the boundaries of a field do not land on byte-boundaries, rational numbers may be supplied as the *start* and *end* values. So most of the time you can pick up a string from (16 32) or an integer from (0 2), but sometimes you would get a boolean from (3/8 4/8) or an integer from (1/2 3/2). An error is signalled if the rational does not specify an integral bit-address.

The alien-field-type argument is a symbol that tells how the field is to be interpreted by the Lisp system. Each alien field type associates a particular alien-format representation with some internal Lisp data-type; functions exist for turning the contents of the field into the internal object and for packing an internal object of the right type back into the field. Some of these types will be built-in:

`string`
: On access, the field is interpreted as a string, one character per byte, and the corresponding Lisp string is returned. The `setf` form accepts a Lisp string and puts the characters into the field.

`perq-string`
: Like `string` except that the first byte in the field is the number of characters, and the remaining bytes are the characters. On access, the length of the Lisp string will be determined by the first byte of the field. The `setf` form will set the first byte according to the length of the Lisp string. The size of the field may not be greater than 256 bytes.

`signed-integer`
: The field is interpreted as a signed, two's complement integer, and the corresponding Lisp integer is returned. On write, the process is reversed.

`unsigned-integer`
: The field is interpreted as an unsigned, positive integer, and the corresponding Lisp integer is returned.

`bit-vector`
: The field is returned as a bit-vector.

`port`
: The field contains an Accent IPC port specified as a 32-bit integer. The internal Lisp format is a `port` structure, with the integer value hidden inside.

`(selection` *s0 s1 s2*...`)`
: The *Sn* are evaluated (at access or `setf` time) to produce arbitrary Lisp objects. On access the alien field is interpreted as an unsigned integer, and the corresponding *Sn* value is returned inside the Lisp. On output, the setting function receives one of the values and stores the corresponding integer into the field. Comparison of items against *Sn* values is done with `eql`.

`ieee-single-flonum`
: The field is interpreted as containing a 32-bit IEEE-format flonum, and this is returned as the internal Lisp flonum type that most closely matches this type. This will vary from one implementation to another, but will be constant within a given implementation. An

implementation would provide whatever floating formats are important in its host environment -- the VAX might provide D, F, G, and H formats rather than IEEE formats.

In each case, the `setf` form will signal an error if the specified field is too small to hold the item coming from Lisp. For integers, the error will occur if significant bits would be lost in doing the write.

6.2. Defining Other Field Types

In addition to these built-in primitive alien field types, the user can define his own via `def-alien-field-type`, a macro with the following arguments:

```
(def-alien-field-type name internal-type primitive-type
                      access-fn setf-fn)
```

internal-type is a Common Lisp type specifier indicating the type of internal Lisp object that the field will be mapped to. *primitive* is any pre-defined alien-field-type: one of the primitives defined above or a field type defined earlier by `def-alien-field-type`. On access, this is applied to the alien object to extract a Lisp object; then this object is passed to the access function, usually a function of one argument, for further processing. For a `setf`, the new value is first passed through the `setf` function, also usually a function of one argument; the result of this is then packed into the alien structure as indicated by *primitive-type.*

For example, suppose we wanted to create a new field-type named `backwards-string`, in which the alien field is treated as a reversed string. This would be done as follows:

```
(def-alien-field-type reverse-string
  'string                        ; Turns to string internally.
  'string                        ; Primitive access to get a string.
  #'(lambda (x) (reverse x))     ; Reverse string on access.
  #'(lambda (x) (reverse x)))    ; Also reverse string on setf.
```

Once this is done, the *reverse-string* field type can be used in `def-alien-structure-type` and as a primitive in defining still more complex field types.

Sometimes, it is desirable to create an alien field type in which the access and `setf` conversion functions can take additional parameters. The *selection* field-type, listed above among the primitives, is one such type. To achieve this effect, one defines an alien field type whose access and `setf` functions take more than one argument. The additional arguments should be optional. When a field-type expression in `def-alien-field` is a list rather than a symbol, the `car` of the list is the type name, and the remaining elements are expressions which are evaluated at access and `setf` time. The results of these evaluations are passed to the access and `setf` functions as additional arguments. This all sounds more complex than it really is. To produce the *selection* field type, if this were not built in as a primitive, one would do the following:

```
(def-alien-field-type selection
  't                             ; Produces any kind of Lisp object.
  'unsigned-integer              ; Primitive access as unsigned integer.
  #'(lambda (n &rest s-list)
      (nth n s-list))            ; Select Nth value in list of choices.
  #'(lambda (x &rest s-list)
      (position x s-list)))      ; Find index of item in list, EQL test.
```

6.3. Variable-Format Structures

The machinery described above is optimized for dealing with alien structure types whose fields are fixed in size and position at the time the structure-type is defined. Given the nature of software outside the Lisp world, this is the sort of thing we will be seeing the most of. However, it would also be nice to be able to use the alien field type translation for packing and unpacking variable-format records. To handle this variable case without getting too complicated, the following simple packing and unpacking macros are provided:

`(alien-field alien-structure alien-field-type` *start end*`)`

This form can be used to access an arbitrary field in any type of alien structure, using the specified alien-field-type, *start*, and *end*. This form pays no attention to any fixed-position fields that may have been defined for structures of this type; it just does what you tell it to do. The alien-field-type may be any pre-defined type. The *start* and *end* arguments are expressed in terms of 8-bit bytes, and may be ratios if it is necessary to reference a field that does not lie on even byte boundaries. This form may be used within a `setf` to alter a field.

```
(pack-alien-structure name length
  (value alien-field-type start end)
  (value alien-field-type start end)
  ...)
```

This creates and returns a new alien-structure object with the specified name and length (in bytes). The name may or may not be an alien-structure-type that has already been defined; in any event, the name is simply stored away and has no effect on how this object is filled. Each of the values is evaluated (to produce a Lisp object) and then is packed into the specified place in the new alien structure using the `setf`-transform of the specified alien field type. For example, to send a variable-length string from Lisp to wherever, something like the following function might be employed:

```
(defun export-string (s)
  (send-external-message wherever
    (pack-alien-structure 'string-message (+ (length s) 4)
      ((length s) 'unsigned-integer 0 4)
      (s 'string 4 (+ 4 (length s))))))
```

To receive a string of the same format:

```
(defun import-string ()
  (let* ((foo (receive-external-message wherever))
         (string-length (alien-field foo 'integer 0 4)))
    (alien-field foo 'string 4 (+ string-length 4))))
```

Index

Index of Concepts

Index of Variables

Index of Constants

Index of Keywords

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Index of Functions, Macros, and Special Forms

Table of Contents

List of Tables

Printer Ruby

Spruce version 12.0 -- spooler version 12.0

File: PRVA:<SLISP.SLM>CLMWIZ.MSS.3

Creation date: 23-Nov-83 12:36 (Wed p.m.)

Printing date: 23-Nov-83 13:00:46 EST

For: STEELE on CMU-CS-C

20 total sheets = 19 pages, 1 copy.

Index

Table of Contents