CARNEGIE-MELLON UNIVERSITY

DEPARTMENT OF COMPUTER SCIENCE

User's Guide for CMU Common Lisp on the IBM RT PC under Mach

Edited by David B. McDonald

16 October 1986

Companion to *Common Lisp: The Language*

Supported by the Defense Advanced Research Projects Agency, Department of Defense, ARPA Order 3597, monitored by the Air Force Avionics Laboratory under contract F33615-81-K-1539. 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.

Table of Contents

Acknowledgements

This manual is a modified version of *Spice Lisp User's Guide* edited by Scott E. Fahlman and Monica J. Cellio. It has been updated to reflect differences between the Common Lisp implementation on the Perq and the IBM RT PC.

Chapter 1

Introduction

CMU Common Lisp is the implementation of Common Lisp for the IBM RT PC running the Mach operating system. It is adapted from Spice Lisp, a Common Lisp implementation developed by CMU's Computer Science Department for the Perq Workstation. All the code for CMU Common Lisp is in the public domain.

The central document for users of any Common Lisp implementation is *Common Lisp: The Language*, 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 facilities. This document covers those choices and features that are specific to the CMU Common Lisp implementation on the IBM PC RT for the Mach operating system. *Common Lisp: The Language* and *User's Guide for CMU Common Lisp on the IBM RT PC under Mach*, taken together, should provide everything that the user of Mach RT PC Common Lisp needs to know.

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

Mach RT PC Common 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 CMU Common Lisp at CMU should watch the Mach, Unix, and CLISP bulletin boards for release announcements, pointers to updated documentation files, and other information of interest to the user community.

1.1. Obtaining and Running Mach RT PC Common Lisp

In order to run Mach RT PC Common Lisp, you must have an IBM RT PC with at least 4 megabytes of memory and a floating point accelerator card. To use Hemlock, you currently also need an IBM AED (Viking), IBM 6155 (APA16), or IBM 6153 (APA8) display. At CMU, there is a misc collection named `rtlisp` which should be updated on your machine regularly by normal mechanisms. For those outside of CMU, there are two files: lisp and lisp.core that need to be installed. Lisp is a small C program that loads lisp.core into memory. Lisp should be put in any bin directory that is normally in your search path. Lisp currently expects to find lisp.core in the directory /usr/misc/.lisp/lib.

To run Lisp, just type:

```
lisp
```

This will start up Lisp with the default core image (/usr/misc/.lisp/lib/lisp.core) after several seconds. Currently Lisp accepts the following flags:

-c Immediately following this flag should be the name of a core file. Rather than using the default core file, the core file specified is loaded.

The Hemlock spelling dictionary is kept in file `spelldict`.

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 CMU Common Lisp on the IBM RT PC for Mach.

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 mantissa, the "hidden bit". The long-float representation conforms to the 64-bit IEEE standard, except that we do not support all the exceptions, negative 0, infinities, 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

CMU Common Lisp characters have 8 bits of `code`, 8 bits of `font`, and 8 `control` bits. The four least-significant bits are named `Control`, `Meta`, `Super`, and `Hyper`, as described in the COMMON LISP manual. Characters read from a normal file or terminal stream always have zero font and bits, but programs can use them 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, CMU Common Lisp has separate packages `hemlock` and `hemlock-internals` (for the editor), `compiler` and `debug`, as well as a large number of packages created for matchmaker interfaces.

2.5. The Editor

The `ed` function will invoke the Hemlock Editor. Hemlock is described in *The Hemlock User's Manual* and *The Hemlock Command Implementors Manual*; like CMU Common Lisp, it contains easily accessible internal documentation.

2.6. Time Functions

The standard COMMON LISP time functions are available in CMU Common Lisp, but no additional facilities such as time parsing and printing are available.

`time` *form* [*Macro*]

The `time` macro evaluates its single form argument, prints the total *elapsed* time for the evaluation to `*trace-output*`, and returns the value which form returns.

`internal-time-units-per-second` [*Constant*]

The internal time unit is one microsecond.

2.7. Garbage Collection

The following two variables control the behavior of garbage collection.

`*gc-trigger-threshold*` [*Variable*]

CMU Common Lisp automatically does a GC whenever the amount of memory allocated to dynamic object exceeds the value of the variable `*gc-trigger-threshold*` (in bytes), unless garbage collection is inhibited. The default value is 4000000.

`*gc-reclaim-goal*` [*Variable*]

If `*gc-reclaim-goal*` bytes are not reclaimed, then `*gc-trigger-threshold*` is increased by the difference between `*gc-reclaim-goal*` and what was reclaimed. The default value is 4000000.

Note that a garbage collection will not happen at exactly `*gc-trigger-threshold*` bytes. The system periodically checks whether `*gc-trigger-threshold*` has been exceeded, and only then does a garbage collection. Automatic garbage collection can be turned off using the `gc-off` function, and turned back on using the `gc-on` function.

2.8. Describe

In addition to the basic function described below, there are a number of switches and other things that can be used to control `describe`'s behavior.

`describe` *object* `&optional` *stream* [*Function*]

The `describe` function prints useful information about *object* on *stream*, which defaults to `*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:

`hash-table` `describe` prints the number of entries currently in the hash table and the number of buckets currently allocated.

`function` `describe` prints a list of the function's name (if any) and its formal parameters. If the name has documentation, then the documentation string will be printed. If the function is compiled then the file where it is defined will be printed as well.

`fixnum` `describe` prints whether the integer is prime or not.

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

If there is anything interesting to be said about some component of the object, describe will invoke itself recursively to describe that object. The level of recursion is indicated by indented output.

`*describe-level*` [*Variable*]

The maximum level of recursive description allowed. Initially two.

`*describe-indentation*` [*Variable*]

The number of spaces to indent for each level of recursive description, initially three.

`*describe-verbose*` [*Variable*]

If true, more information will be printed than usually would be. Initially `nil`.

`*describe-print-level*` [*Variable*]
`*describe-print-length*` [*Variable*]

The values of `*print-level*` and `*print-length*` during description. Initially two and five.

`*describe-implementation-details*` [*Variable*]

If true `describe` will print out everything there is, otherwise information which is internal to the implementation is not printed. This currently controls display of various properties.

`defdescribe` *function-name type lambda-list* {*form*}* [*Macro*]

This macro is used to tell describe about new types. It creates a function called *function-name* with the specified *forms* and *lambda-list* which is called with the object when `describe` is asked to describe an object of the specified *type*. Output should be directed to `*standard-output*`. The code may call `describe`, in which case it will do the right thing. Users are encouraged to observe the values of `*describe-verbose*` and `*describe-implementation-details*` where appropriate.

If *type* is symbol, then the second and third values returned by the body are interpreted as lists of property names and kinds of documentation effectively used up by the `defdescribe` method.

Returning these values inhibits the default action of displaying the specified documentation or property.

2.9. Modules

The CMU Common Lisp implementation of modules operate as described in the Common Lisp manual. In addition, the following things are also true.

When the user requires a module, the system initially looks in the file named in *slisp-modules-file* (slisp-modules.slisp, by default) on the current path. This file should contain an a-list mapping module names to lists of files which should be loaded when those modules are required. This a-list is put on the variable *module-file-translations* when it is read, and once this variable has a useful value the file is not read anymore. If this list of files cannot be found, either in the file or the a-list, then the file whose name is the same as that of the module is loaded.

If *require-verbose* is non-nil (the default) require prints out the name of each file that it looks at and tells whether it loaded said file or not.

2.10. Saving a Core Image

A mechanism has been provided to save a running Lisp core image and to latter restore it. This is convenient if you don't want to load several files into a Lisp when you first start it up.

`save` *file* `&optional` *(checksum t)* [*Function*]

The `save` function saves the state of the currently running lisp core image in *file*. Currently the *checksum* argument is ignored.

To resume a saved file, type:

```
lisp -c file
```

2.11. Interrupts

Under Mach, an interrupt capability is enabled. CMU Common Lisp responds to various Unix signals in a non-standard way:

SIGINT causes Lisp to enter a break loop. This puts you into the debugger which allows you to look around at the current state of the computation. If you proceed from the break loop, the computation will be restarted where it was interrupted. This signal can be generated from the keyboard by typing control-B or function key F1. While in Hemlock, only F1 has this meaning.

SIGQUIT causes Lisp to do a throw to the top-level. This causes the current computation to be aborted, and control returned to the top-level read-eval-print loop. This signal can be generated from the keyboard by typing control-G or function key F2. While in Hemlock, only F2 has this meaning.

SIGTSTP causes Lisp to suspend execution and return to the Unix shell. If control is returned to Lisp, the computation will proceed from where it was interrupted. This signal can be generated from the keyboard by type control-Z or function key F3. While in Hemlock, only F3 has this meaning.

When a signal is generated from the keyboard, there may be some delay before it is processed since Lisp cannot be interrupted safely in an arbitrary place. The computation will continue until a safe point is reached and then the interrupt will be processed.

Other Unix signals that correspond to program errors cause the Lisp error system to obtain control. Under normal circumstances this should not happen, but if it does and you have important work, you should immediately try to save it.

2.12. Search Lists

Search lists make it possible to refer to files using abbreviated names. The general form of a search list definition is:

```
(setf (search-list "name:") '(directory1 directory2 ...))
```

Where name is the name of the search list, and $directory_i$ are strings that specify Unix directories. For example, it is possible to define the search list code: as follows:

```
(setf (search-list "code:") '("/usr/lisp/code/"))
```

It is now possible to use code: as an abbreviation for the directory /usr/lisp/code/ in all file operations. For example, you can now specify code:eval.slisp to refer to the file /usr/lisp/code/eval.slisp.

To obtain the value of a search-list name, use the function search-list as follows:

```
(search-list "name:")
```

Where name is the name of a search list. If name is not defined as a search-list NIL is returned. For example, calling search-list on code: as follows:

```
(search-list "code:")
```

returns the list ("/usr/lisp/code/").

Chapter 3

Debugging Tools

By Jim Large and Steve Handerson

3.1. 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 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

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

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` This is used as the combination of `: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-print-level*` [*Variable*]

`*trace-print-length*` [*Variable*]

`*print-level*` and `*print-length*` are bound to `*trace-print-level*` and `*trace-print-length*` when printing trace output. The forms printed by the `:print` options are also affected. `*Trace-print-level*` and `*trace-print-length*` 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.

3.1.1. 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

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

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`.

3.2. The Single Stepper

`step` *form* [*Function*]

Evaluates form with single stepping enabled or if *form* is `T`, enables stepping on 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 `*print-level*` and `*print-length*` bound to `*step-print-level*` and `*step-print-length*`. 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-print-level*` or `*step-print-length*`.)
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 the current call to eval.
g	Throw to top level.

`*step-print-level*` [*Variable*]
`*step-print-length*` [*Variable*]

`*print-level*` and `*print-length*` are bound to these values when the current form is printed. `*Step-print-level*` and `*step-print-length*` 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. Initially set to some reasonable number.

3.3. 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.

Inside the debugger, most commands refer to the *current stack frame.*[3] The debugger assigns numbers to the frames on the stack, starting with zero as the most recent and increasing deeper into the stack. The debug prompt includes the number of the current frame as its main feature.

Most expressions typed to debug are simply evaluated as they would have been had you not entered debug. This includes the special debugger functions to be described, which are meaningful only inside the debugger. The biggest exception is the debugger commands, which are either one or two letters. These may display information about the current frame or change the current frame, but they generally do not affect the evaluation history (*, **, and friends).

3.3.1. 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. Frames which are not active are marked with a "*", and the reconstructed call consists of what arguments are present on the stack. `*Debug-print-level*` and `*debug-print-length*` affect the style of the printing.

---

[3] The debugging functions may refer to other frames by number. This will be described shortly.

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

u Move up to the next higher visible frame. More recent function calls are considered to be higher on the stack.

d Move down to the next lower visible frame.

t Move to the highest visible frame.

b Move to the lowest visible frame.

f Move to a given frame, visible or not. Prompts for the number.

3.3.2. 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 `*print-level*` and `*print-length*` instead of `*debug-print-level*` and `*debug-print-length*`.

(debug-value *symbol* [*frame*])
Returns the value of *symbol*, considered as a special variable, in the binding context of either the current frame, or the numbered frame, if specified.

(debug-local *n* [*Frame*])
Returns the value of the *n*th local variable in the current or specified frame.

(debug-arg *n* [*frame*])
Returns the *n*th argument of the frame.

(debug-pc [*frame*]) Returns the next instruction to be executed in the specified (active) frame. Can be used with Disassemble.

3.3.3. Other Commands

h Prints a brief but comprehensive list of commands on the terminal.

q Causes `debug` to return `nil`.

(**debug-return** *expression* [*frame*])
Forces the current function to return zero or more values. If the function was not called for multiple values, only the first value will be returned.

(**backtrace**) Prints a history of function calls. The printing is controlled by `*debug-print-level*` and `*debug-print-length*`. Only those frames which are considered visible by the frame movement commands will be shown.

(**debug-hide** *option* [*arg(s)*])
Makes the described stack frames invisible to the frame movement commands. The second argument is evaluated and may be a symbol or a list; the function returns the hidden members of the category. With no arguments, returns the current filter (hidden frames). *option* is a subcommand which may be one of:

`package(s)` Calls to hidden packages are visible, but calls within them are not.

`function(s)` Calls to the named functions will not be visible.

`type(s)` Hides miscellaneous frame and function types, any of:

- `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.

`open` Open frames will not be visible.

`active` Active frames will not be visible.

`catch` Catch frames will not be visible.

(**debug-show** *options arg or args*)
Cancels the effect of the corresponding **debug-hide**. Note that a frame may be hidden in a variety of other ways, though.

`debug` [*Function*]
Invokes the debugger. `debug` always returns `nil`.

`*debug-print-level*` [*Variable*]
`*debug-print-length*` [*Variable*]
`*print-level*` and `*print-length*` are bound to these values during the execution of some debug commands. When evaluating arbitrary expressions in the debugger, the normal `*print-level*` and `*print-length*` 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**.

3.4. 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`.

`(debug)` Enter the debugger. This can be done anywhere, but it is usually done from a break loop.

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.

3.4.1. 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 effects 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 evaled for side effects. 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>
  .
  .
  .)
```

Chapter 4

The Compiler

4.1. Calling the Compiler

Functions may be compiled using `compile`, `compile-file`, or `compile-from-stream`. `Compile` operates exactly as documented in the *Common Lisp Reference Manual.*

`compile-file &optional` *input-pathname* `&key :output-file :error-file :lap-file :errors-to-terminal :load` [*Function*]

This function is an expanded version of that described in the *Common Lisp Reference Manual.* If `input-pathname` is not provided `compile-file` prompts for it. `Output-file` and `Error-file` default to `T`, producing a fasl file and a compilation log with extensions .sfasl and .err. `Lap-file` defaults to `nil`, indicating that the lap code should not be stored in a file. Any of these options may be `t`, `nil`, or the string name of a file to write to. *Errors-to-terminal* defaults to `T`; if specified and `nil` the compilation log goes only to the .err file. If `load` is specified and non-`nil` the compiled file is loaded after the compilation.

`compile-from-stream input-stream` [*Function*]

This function takes a stream as an input and reads lisp code from that stream until end of file is reached. The code is compiled and loaded into the current environment. No output files are produced.

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`.

`*optimize-let-bindings*`
If this is t, optimize some let bindings, such as those generated by lambda expansions and setf based operations. If it is :all, optimize all lets. If it is nil, don't optimize any. It takes significant time to do all. The optimization involves replacing instances of variables that are bound to other variables with the other variables. Defaults to `t`.

`*examine-environment-function-information*`
If this is non-NIL, look in the compiler environment for function argument counts and types (macro, function, or special form) if you don't get the information from declarations. Defaults to `t`.

`*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 iteration. 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 Maclachlan

In CMU Common Lisp, 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 CMU Common Lisp, compiled code typically runs at least 100 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 innocuous 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

Before using an association list (alist) or a symbol property, you should consider whether a hash-table would do the job better. There are two arguments: efficiency and style.

Since `assoc` is implemented directly in assembler code when the *test* argument is `eq` or `eql`, it is fairly fast when there are only a few elements, but the time goes up in proportion with the number of elements. In contrast, the hash-table lookup has a somewhat higher overhead, since a function call is involved, but the speed is largely unaffected by the number of entries in the table. The following table shows the number of microseconds it takes to do a failing lookup in a `eq` hash-table and an alist, where *n* is the number of entries:

n	*hashtable*	*alist*
10	490	281
100	520	2141
1000	520	22870

As you can see, the break-even point is between ten and twenty. For an `equal` hash-table or alist, hash-tables have an even greater advantage, since the test is more expensive and the alist lookup is not done in assembler code. Whatever you do, be sure to use the most restrictive test function possible.

The style argument observes that although hash-tables and alists overlap in function, they do not do all things equally well.

- Alists are good for maintaining scoped environments. They were originally invented to implement scoping in the Lisp interpreter, and are still used for this in CMU Common Lisp. With an alist one can non-destructively change an association simply by consing a new element on the front. This is something that cannot be done with hash-tables.

- Hashtables are good for maintaining a global association. The value associated with an entry can easily be changed by doing a setf. With an alist, one has to do go through contortions, either `rplacd`'ing the cons if the entry exists, or pushing a new one if it doesn't. The side-effecting nature of hash-table operations is an advantage here.

Experienced Lisp programmers will notice that I am suggesting that hash-tables be used for things which symbol properties are often used for. There are a number of reasons to use hash-tables instead of properties:

- Hash-tables can be more efficient if the average property list length is sufficiently large.
- A hash-table is inherently anonymous, while a property is usually a symbol. A new set of associations can be created simply by making a new hash-table. A similar effect could be obtained by using gensyms as property names, but this is apt to cause nausea.
- A hash-table is one object rather than a bunch of stuff scattered across dozens of property lists. This means that modularity is improved and bugs find it harder to propagate.

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 to improve performance. 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. Using a bit-vector will nearly always be faster, and can be tremendously faster if the number of elements in the set is not small. The logical operations on *simple* bit vectors are implemented in assembler code.

5.5. Simple Vs Complex Arrays

If an array is a `simple-string`, `simple-vector` or `simple-bit-vector`, more efficient code if the compiler is told the type. *Declare Your Vector Variables*—If you don't the compiler will be forced to make worst-case assumptions. Example:

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

Arrays with more than two dimensions are accessed by Lisp code, thus accessing any such array is many times slower than accessing a vector or two-dimensional array.

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. CMU Common Lisp on the IBM RT PC for Mach is fairly successful in this respect. *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.

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 CMU Common Lisp. The main problem with rest args is that the assembler code must cons a list to hold the arguments. If a function is called many times or with many arguments, large amounts of consing will 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.

A 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. 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.

5.8. Numbers

CMU Common Lisp provides six types of numbers for your enjoyment: fixnums, bignums, ratios, short-floats, long-floats and complexes. 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. Ratio and complex arithmetic are implemented in Lisp rather than assembler; this results in orders of magnitude slower execution.

5.9. Timing

The first step in improving a program's performance is to make extensive timings to find code which is time-critical. The `time` macro is the best way currently available to do timings. For things which execute fairly quickly it may be wise to time more than once, since there may be paging overhead in the first timing. The times that `time` gets are only accurate to a certain number of decimal places, so for small pieces of code it may be a good idea to write a *compiled* driver function which calls the function to be tested a few hundred times. If one finds the time and divides by the number of iterations, then fairly accurate statistics can be collected. Be very careful when using `get-internal-run-time`, since it takes a substantial amount of time to execute.

Chapter 6

The Alien Facility

By Rob Maclachlan

6.1. What the Alien Facility Is

Aliens provide a mechanism in Lisp for manipulating objects which are sent and received in Accent IPC messages. Normally Aliens are used to implement a Matchmaker remote procedure call message interface. In order to produce a Lisp message interface, Matchmaker analyzes the shared definitions file and produces Lisp code written primarily in terms of the functions, macros and special forms defined by the Alien facility.

6.2. Alien Values

Objects in messages are manipulated via typed pointers to the data involved. These typed pointers are called *Alien values.* An Alien value is a Lisp object which consists of three components:

address — The address of the object pointed to. This is a word address, which may in general be a ratio, since objects need not be word aligned.

size — The size in bits of the object pointed to. This information is used to make sure that accesses to the object fall within it.

type — The Alien type of the object pointed to. Since Alien values have a type, functions that use them can check that their arguments are of the correct type.

6.3. Alien Types

Alien types are tags attached to Alien values that may be checked to assure that they are not used inappropriately. When types are compared the comparison is done with the Lisp `equal` function. Types are typically represented by symbols or lists of symbols such as the following:

```
string
(directory-entry type-file)
(signed-byte 7)
string-char
```

A convention which is encouraged, but not enforced, is that an ordinary type is represented by a symbol, and a type with some subtype information, such as a discriminated union is represented as a list of the main type and the subtype information.

6.4. Alien Primitives

This section describes the defined Alien primitives. Some of these primitives are intended to be used only in code generated by matchmaker, while others might be used by mere mortals.

`make-alien` *type size* `&optional` *address* [*Function*]

Make an Alien object of type *type* that is *size* bits long. *address* may be either a number, `:static` or `:dynamic`. If address is a number, then that becomes the returned alien's address. If *address* is `:static` or `:dynamic` then storage is allocated to hold the data. Aliens that are allocated statically are packed as many as will fit on a page, resulting in increased storage efficiency, but disallowing the deallocation of the storage. Since static aliens are allocated contiguously, the `save` function can arrange to save their contents, permitting initialization of such Aliens to be done only once. Dynamic Aliens are allocated on page boundaries, and may be deallocated using `dispose-alien`.

`alien-type` *alien* [*Function*]
`alien-size` *alien* [*Function*]
`alien-address` *alien* [*Function*]

These functions return the type, size and address of *alien*, respectively.

`alien-sap` *alien* [*Function*]

This function returns the address of *alien* as a system-area-pointer. If the address is not an integer, an error will be signaled, since it cannot be represented as a system-area-pointer.

`copy-alien` *alien* [*Function*]

Copy the storage pointed to by *alien* and return a new Alien value that describes it.

`alien-assign` *to-alien from-alien* [*Function*]

Copies the bits in *from-alien* into *to-alien*. The alien values must be of the same size and type.

`dispose-alien` *alien* [*Function*]

Release any storage associated with *alien*. Any reference to *alien* afterward may lose horribly.

`alien-access` *alien* `&optional` *lisp-type* [*Function*]

`alien-access` returns the object described by *alien* as a Lisp object of type *lisp-type*. An error is signalled if the type of *alien* cannot be converted to the given *lisp-type*. For most lisp-types the corresponding Alien type is identical. If the Lisp type is uniquely determined by the type of the *alien* then *lisp-type* need not be supplied.

lisp-type must be one of the following types:

`(unsigned-byte` *n*`)`
An unsigned integer *n* bits wide, as in COMMON LISP.

`(signed-byte` *n*`)`
A signed integer *n* bits wide.

`boolean` A one bit value, represented in Lisp as `t` or `nil`.

`(enumeration` *name*`)`
Access a value of the enumeration *name*. Enumerations are defined by the macro `defenumeration` (page 28).

`string-char` An eight-bit ASCII character.

`simple-string` The corresponding Alien type is `perq-string` which is a Perq Pascal string.

`port` An Accent IPC port.

`short-float long-float`
There is only one Alien type accepted by these, `ieee-single` which is a floating point number in pseudo IEEE single format, as used by Perq Pascal. Both lisp-types are allowed so that one may choose whether to cons long-floats or lose precision. Note however, that no floating-point type is implemented---the purpose of this entry is solely to confuse anyone who has read this far.

`system-area-pointer`
Return as a system-area-pointer the long-word described by *alien*. It is an error for the address not to be in the system area. This lisp type may also be used with the `alien` alien type.

If `alien-access` is set with `setf` then the inverse type conversion is done, and the alien set to the new value. When setting, an additional type is available:

`(pointer` *type*`)` *type* may be any unboxed Lisp type such as `simple-string`, `simple-bit-vector` and `(simple-array (unsigned-byte 8))`. When an object of such a type is stored the address of the first data word is stored in the corresponding location.

`(alien` *type* [*size*]`)`
This lisp type is used to access a pointer as an alien value. When read, an alien value created out of the pointer, *type* and *size* is returned. When set, the address of the alien values is written. When read, the *size* must be specified, when set it is ignored.

`defenumeration` *name* $\{\{element\}^+ \mid \{(element\ value)\}^+\}^*$ [*Macro*]

Define an enumeration type for use with `alien-access`. The enumeration may be used with the `enumeration` Alien type by specifying its *name*. Each successive *element* is assigned a numeric value, starting at zero. Each element must be a keyword symbol. Example:

```
(defenumeration era :stone-age :medieval :now :space-age)

(alien-store (language-era (alien-value pascal))
             (enumeration era)
             :stone-age)
```

The numeric value for an element may be specified by using a list of the keyword and the numeric value. If the value is specified for any element then it must be specified for all. Each value must be

an integer.

```
(defenumeration silly (:a -32) (:b 15) (:c 1000000))
```

6.5. Alien Variables

An Alien variable is a symbol that has had an Alien value associated with it. An Alien variable is not a Lisp variable -- in order to obtain the value of an Alien variable, the special form `alien-value` must be used. The reason for using Alien variables as opposed to Lisp variables is that various additional information can be associated with the Alien variable which may permit code which refers to it to be compiled more efficiently.

`alien-value` *name* [*Special form*]

Return the value of the Alien variable *name*.

`alien-bind` ({(*name value type* [*aligned*])}*) {*form*}* [*Special form*]

`alien-bind` defines a local Alien variable *name* having the specified Alien *value*. Bindings are done serially, as by `let*`. If *aligned* is supplied and true, then the *value* is asserted to be word aligned. Hopefully this feature will be replaced with something less silly.

`defalien` *name type size* [*address*] [*Macro*]

Defines *name* as an Alien variable, creating a value from *type*, *size* and *address* as for `make-alien` (page 27). *name* and *type* are not evaluated. Since the alien-value for a defalien created variable is kept in the value cell of the symbol it is not necessary (but legal) to use `alien-value` to obtain the value.

6.6. Alien Stacks

For some purposes it is useful to have stack allocation of Alien values. Alien stacks are used by Matchmaker to receive messages into, since a software interrupt may cause an interface to be entered recursively.

`define-alien-stack` *name type size* [*Macro*]

Defines a stack of static Aliens having the specified *type* and *size*. The stack has no maximum size, since new Aliens are allocated whenever they are needed.

`with-stack-alien` (*var name*) {*form*}* [*Special form*]

Binds the Alien variable *var* to an Alien value from the Alien stack with the specified *name* during the evaluation of the *forms*.

6.7. Alien Operators

An Alien operator is a function which returns an Alien value. When an Alien operator is defined via the `defoperator` macro, the type of the result and all of the Alien valued arguments is specified. If an argument to an Alien operator is not the of the correct type an error is signalled. Because of the way an Alien

operator is specified, it can be compiled much more efficiently than a function that does the same thing.

`defoperator` (*name result-type*) ({(*arg arg-type*) | *arg*}*) [*doc-string*] *body* [*Macro*]

This macro defines *name* as an Alien operator returning a value of type *result-type*. *doc-string*, if supplied, becomes the function documentation for the function created.

The *args* to the operator are similar to the binding specifiers to `alien-bind` (page 29). If the type of the argument is specified, then the argument must be an Alien value of the specified type, otherwise it may be any Lisp value.

`defoperator` is similar to the complex form of `defsetf` or `defmacro` in that the body is evaluated at compile time, the result of the evaluation being the desired code. When the body is evaluated, Lisp variables having the arguments' names are bound to markers which must appear in the resulting code where a reference to that argument is desired. Normally the form which results from the evaluation of the body consists solely of combinations of `alien-index` and `alien-indirect` on arguments and simple numeric functions thereof.

`alien-index` *alien offset size* [*Function*]

This function indexes into *alien* by *offset* bits and returns an Alien value *size* bits long. It is an error for the field so selected not to fit inside *alien*. Normally this function is used only within the definition of an Alien operator, so the type of the resulting value is `nil` to indicate that it has no particular type

`alien-indirect` *alien size* [*Function*]

This function takes two words at the place described by *alien* and treats them as a pointer, returning a new Alien value which describes the piece of memory pointed to by that pointer which is *size* bits long. It is an error for *alien* not to describe a piece of storage suitable for use as a pointer. Like `alien-index`, this is normally only used within the definition of an Alien operator, and its result type is `nil`.

`long-words` *n* [*Function*]
`words` *n* [*Function*]
`bytes` *n* [*Function*]
`bits` *n* [*Function*]

These functions are equivalent to multiplication by thirty-two, sixteen, eight and one respectively. They also assert their argument to be an integer. Use of these function in the definition of Alien operators can make the definition clearer, and give additional information that can be used to produce better compiled code.

6.8. Examples

This Pascal declaration might be translated into the following Alien operator definitions:

```
Foo = RECORD
          A: Integer;
          B: ^Array [0..99] OF ^Foo;
      END;
```

```
VAR
        F: Foo;

 <==>

;;; This operator selects the A field from a Foo.  The type of the
;;; resulting Alien is (signed-byte 16), which is what a Perq Pascal
;;; integer is.  It takes one argument called Foo which is an Alien value
;;; of type Foo.  Since A is the first field in the record, we index into
;;; the Alien by zero bits.  The size of the result is sixteen bits, or one
;;; word.  Alien-Value must be used on the parameter, since it is an
;;; Alien variable.
;;;
(defoperator (foo-a (signed-byte 16)) ((foo foo))
  '(alien-index (alien-value ,foo) 0 (words 1)))

;;; This operator extracts the B field from a Foo.  The result type is
;;; (ref (array (ref foo) 100)), indicating that it is a pointer to an
;;; array of pointers to foos.  Note the use of list Alien types to
;;; indicate subtype information, but remember that this is merely a
;;; convention.  The B field is one word into the record, and since it is a
;;; pointer, it is thirty-two bits, or one long-word long.
;;;
(defoperator (foo-b (ref (array (ref foo) 100))) ((foo foo))
  '(alien-index (alien-value ,foo) (words 1) (long-words 1)))

;;; This operator dereferences a pointer to an (array (ref foo) 100).  The
;;; size of the resulting Alien is one hundred long-words, since the array
;;; contains one hundred thirty-two bit pointers
;;;
(defoperator (deref-array-ref-foo-100 (array (ref foo) 100))
             ((ra (ref (array (ref foo) 100))))
  '(alien-indirect (alien-value ,ra) (long-words 100)))

;;; Index into an (array (ref foo) 100).  Here we have a non-alien-valued
;;; parameter I, which is the index into the array.
;;;
(defoperator (index-array-ref-foo-100 (ref foo))
             ((a (array (ref foo) 100)) i)
  '(alien-index (alien-value ,a) (long-words ,i) (long-words 1)))

;;; Dereference a pointer to a foo.  A foo is three words long.
;;;
(defoperator (deref-foo foo) ((rfoo (ref foo)))
  '(alien-indirect (alien-value ,rfoo) (words 3)))

;;; Define F as an Alien variable, whose type is foo and is three words
;;; long.  Storage to hold the foo will be allocated.
;;;
(defalien f foo (words 3))
```

With this definition, the following Pascal expression could be translated in this way:

```
F.B^[7]^.A

 <==>

(alien-access
 (foo-a (deref-foo (index-array-ref-foo-100
                    (deref-array-ref-foo-100 (foo-b (alien-value f)))
                    7))))
```

If instead of getting the A out of the seventh foo, we wanted a vector containing the first F.A foos in the array F.B↑, we could do this:

```
;; Find how many foos to use by getting the A field.
(let* ((num (alien-access (foo-a (alien-value f))))
       (result (make-array num)))
  ;;
  ;; Bind the Alien value for the array so we don't have to keep
  ;; recomputing it.
  (alien-bind ((a (deref-array-ref-foo-100 (foo-b (alien-value f))))
               (array (ref foo) 100))
    ;;
    ;; Loop over the first N elements and stash them in the result vector.
    (dotimes (i num)
      (setf (svref result i)
            (deref-foo (index-array-ref-foo-100 (alien-value a) i))))
    result))
```

Index