CMU Common Lisp User's Manual
Mach/IBM RT PC Edition

edited by David B. McDonald

September 1987

CMU-CS-87-156

Companion to *Common Lisp: The Language*

Abstract

CMU Common Lisp is an implementation of Common Lisp that currently runs on the IBM RT PC under Mach, a Berkeley Unix 4.3 binary compatible operating system. This document describes the implementation dependent choices made in developing this implementation of Common Lisp. Also, several extensions have been added, including the proposed error system, a stack crawling debugger, a stepper, an interface to Mach system calls, a foreign function call interface, the ability to write assembler language routines, and other features that provide a good environment for developing Lisp code.

This research was sponsored by the Defense Advanced Research Projects Agency (DOD), monitored by the Space and Naval Warfare Systems Command under proposed contract N00039-87-C-0251.

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.

2

Chapter 1

Introduction

CMU Common Lisp is a public-domain implementation of Common Lisp developed in the Computer Science Department of Carnegie Mellon University. Currently, it runs only on the IBM RT PC workstation under CMU's Mach operating system; we may port it to other machines in the future. CMU Common Lisp is descended from Spice Lisp, developed at CMU for the Perq workstation.

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 this User's Guide, taken together, should provide everything that the user of CMU Common Lisp on the IBM RT PC needs to know.

In addition to the language itself, this document describes a number of useful library modules that run in CMU Common Lisp. Hemlock, an Emacs-like text editor is included as an integral part of the CMU Common Lisp environment. It is described in two separate documents: *Hemlock User's Manual* and *Hemlock Command Implementor's Manual*.

Mach and CMU Common Lisp are 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-Announce, Unix-Forum, 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 CMU Common Lisp on the IBM RT PC under Mach

In order to run CMU Common Lisp, you must have an IBM RT PC with at least 4 megabytes of memory and a floating point accelerator card. If you plan to use the X window system at the same time, you should have at least 6 megabytes of memory. The Hemlock editor can be used with the workstation's high-resolution display (the IBM AED (Viking), IBM 6155 (APA16), or IBM 6153 (APA8) are supported) or a standard terminal, such as a Concept-100 or H-19. On a high-resolution display, it is possible to run Hemlock on the bare display or under the X window manager.

At CMU, there is a misc collection named `rtlisp` which should be updated on your machine regularly by normal **`sup`** mechanisms. The standard Mach distribution is set up to request this collection by default. For those outside of CMU, there are five files: lisp, lisp.core, hemlock.cursor, hemlock.mask, and spelldict.bin 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/. If Hemlock is run under the X window system, it needs hemlock.cursor and hemlock.mask. If Hemlock's spelling corrector is used, it needs spelldict.bin. Hemlock expects to find all these files in /usr/misc/.lisp/lib/.

At CMU, you should put either /usr/misc/bin (if you want all the misc executable files) or /usr/misc/.lisp/bin (if you want just Common Lisp) in your PATH searchlist. Typing **lisp** will start up Lisp with the default core image (/usr/misc/.lisp/lib/lisp.core) after several seconds.

Currently Lisp accepts the following switches:

-c	requires an argument that should be the name of a core file. Rather than using the default core file (/usr/misc/.lisp/lib/lisp.core), the core file specified is loaded.
-init	accepts an argument that should be the name of an init file to load during the normal start up sequence. The default is to load init.fasl or, if that does not exist, init.lisp from the user's home directory. If the file is not in the user's home directory, the full path must be specified.
-hinit	accepts an argument that should be the name of the hemlock init file to load the first time the function `ed` is invoked. The default is to load hemlock-init.fasl or, if that does not exist, hemlock-init.lisp from the user's home directory. If the file is not in the user's home directory, the full path must be specified.
-noinit	accepts no arguments and specifies that an init file should not be loaded during the normal start up sequence. Also, this switch suppresses the loading of a hemlock init file when Hemlock is started up with the -edit switch.
-eval	accepts one argument which should be a Lisp form to evaluate during the start up sequence. The value of the form will not be printed unless it is wrapped in a form that does output.
-edit	specifies to enter Hemlock. An optional argument should be the name of the editor Lisp to register with the nameserver. If no argument is given, [machine-name]Editor is used as the name. A file to edit may be specified by placing the name of the file between the program name (usually `lisp`) and the first switch.
-slave	specifies that Lisp should start up as a **slave** Lisp and try to connect to an editor Lisp. The default name for the editor Lisp is [machine-name]Editor. If an optional name is given, the slave Lisp tries to connect to the named editor Lisp.
-register	specifies that a slave Lisp should register with the nameserver. The default name to use is [machine-name]Eval. If an optional name is given, the slave Lisp registers that name with the nameserver.

For more details on the use of the -edit, -slave, and -register switches, see the *Hemlock User's Manual.*

Arguments to the above switches can be specified in one of two ways: <switch>=<value> or <switch><space><value>. For example, to start up the saved core file mylisp.core type either of the following two commands:

```
    lisp -c=mylisp.core
    lisp -c mylisp.core
```

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. As in *Common Lisp: The Language* all symbols and package names are printed in lower case, as a user is likely to type them. Internally, they are normally stored upper case only.

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. There is a round off error of approximately 1 in 10^6 when using short-floats. 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. Since the most positive fixnum is over one hundred million, you shouldn't need to use bignums unless you are counting the reasons to use Lisp instead of Pascal.

2.2. Characters

CMU Common Lisp characters have 8 bits of **`code`**, 8 bits of **`font`**, and 8 **`control`** bits. The four least-significant control 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. All printing functions ignore the font information of a character object. **`Write-char`** and **`princ`** ignore the control bits information; **`print`** and **`prin1`** print a character using **`#\`** notation (e.g., a character with code for A and control and meta bits set prints as **`#\CONTROL-META-A`**). Programs can make use of these fields 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. Defstruct

extensions:*safe-defstruct-accessors* *[Variable]*

Safe-defstruct-accessors controls whether slot accessing code is completely type checked or not. The default value is NIL specifying that completely safe accessors are not created. However, they will still check to make sure all accesses are within range. If the value is non-NIL, the accessor functions check to make sure that the slot accessors are passed a structure of the correct type. This mode is useful when debugging code making use of many structures.

2.5. Packages

When CMU Common Lisp is first started up, the default package is the **user** package. The **user** package uses the **lisp**, **extensions**, **conditions**, **debug**, and **clos** packages. The symbols exported from these five packages can be referenced without package qualifiers.

Currently, the following packages are defined (abbreviations for the packages are in parenthesis after the full name):

lisp	The **lisp** package exports all the symbols defined by the COMMON LISP manual and only those symbols. Strictly portable Lisp code will depend only on the symbols exported from the **lisp** package.
keyword	The **keyword** package contains keywords (e.g., :start). All symbols in the **keyword** package are exported and evaluate to themselves (i.e., the value of the symbol is the symbol itself).
system	The **system** package contains functions and information necessary for the system. This package is used by the **lisp** package and exports several symbols that are necessary to interface to system code. For example, the symbols used by the alien facility are exported from this package.
user	The **user** package is the default package and is where a user's code and data is placed unless otherwise specified. The **user** package exports no symbols.
extensions (ext)	The **extensions** packages exports local extensions to Common Lisp that are documented in this manual. Examples include the **save-lisp** function and the interface to foreign (C) functions.
hemlock (ed)	The **hemlock** package contains all the code to implement Hemlock commands. The **hemlock** package currently exports no symbols.
hemlock-internals (hi)	The **hemlock-internals** package contains code that implements low level primitives and exports those symbols used to write Hemlock commands.
spell	The **spell** package contains a spelling checker and corrector that is used by Hemlock. It exports several symbols that allow a user to manipulate the spelling dictionary and to check the spelling of words.
mach	The **mach** package contains code to interface to the Mach operating system. All the standard unix system calls (the names are unix-<system call name>) and the Mach specific calls (e.g., vm_allocate, port_allocate, etc.) are exported from this package.
debug	The **debug** package contains the stack crawling debugger and the low level functions on which it is built. It exports symbols the user may want to use when debugging a program.
conditions	The **conditions** package contains the new error system as proposed for Common Lisp and exports several symbols necessary for the new error system.
compiler (clc)	The **compiler** package contains the Common Lisp compiler and an assembler for the IBM RT

	PC. This package exports only the symbol assemble-file.
`clos (pcl)`	The `clos` package contains the code that implements the Common Lisp Object System (CLOS) specification and exports the symbols as defined in the CLOS specification. The nickname `pcl` has been retained for compatibility with earlier versions.
`walker`	The `walker` package contains code used by CLOS and exports a few symbols needed by CLOS.
`x`	The `x` package contains an interface to version 10 release 4 of the Xlib C routines using the foreign function calling mechanism described in chapter 10. The `x` package exports all the X C library functions that are necessary to use the X window system from Lisp. It also exports several macro definitions, function definitions, data structures, and constants that make programming X easier.
`mmlispdefs`	The `mmlispdefs` package contains code used by matchmaker generated interfaces such as some functions in the `mach` package. It exports several symbols that the matchmaker generated files need.
`msgn`	The `msgn` package contains matchmaker code to interface to the name server. It exports the functions necessary to access the name server and some constants.
`ts`	The `ts` package contains matchmaker code to implement the server and client side of a typescript interface. This allows lisp to implement a read-eval-print loop that is connected to a remote Lisp.
`tl`	The `tl` package contains matchmaker code to implement the server and client side of the eval server. This allows Lisp expressions to be evaled by a remote Lisp and the results returned to a local Lisp. The remote Lisp may or may not be on the same machine as the local one.
`edit`	The `edit` package contains matchmaker code to allow a Lisp running the eval server to establish a connection to a Lisp running Hemlock.

The `lisp`, `user`, `keyword`, and `system` packages are required by the COMMON LISP manual.

2.6. The Editor

The `ed` function will invoke the Hemlock Editor. Hemlock is described in *Hemlock User's Manual* and *Hemlock Command Implementor's Manual*; like CMU Common Lisp, it contains easily accessible internal documentation. Most user's at CMU prefer to use Hemlock's slave connection or eval mode as the normal way to communicate with Lisp's read-eval-print loop.

2.7. 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.

2.8. Garbage Collection

CMU Common Lisp uses a stop-and-copy garbage collector that compacts the items in dynamic space every time it runs. Most users run GC frequently, long before space is exhausted, in order to compact the working set. With the default values for the following two variables, you can expect a GC to take about one minute of elapsed time on a 6 megabyte machine running X as well as Lisp. On machines with 8 megabytes or more of memory, setting these variables to 1 or 2 million allows a GC to run without much (if any) paging. GC's run more frequently but tend to take only about 5 seconds.

The following three variables control the behavior of garbage collection.

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

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

`extensions:*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.

`extensions:*gc-verbose*` *[Variable]*

If **`*gc-verbose*`** is NIL, no messages will be printed when an automatic garbage collection occurs. Otherwise, a message is printed when a GC starts and another one is printed when a GC completes.

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.

2.9. 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.

extensions:*describe-level* [*Variable*]
The maximum level of recursive description allowed. Initially two.

extensions:*describe-indentation* [*Variable*]
The number of spaces to indent for each level of recursive description, initially three.

extensions:*describe-verbose* [*Variable*]
If true, more information will be printed than usually would be. Initially `nil`.

extensions:*describe-print-level* [*Variable*]
extensions:*describe-print-length* [*Variable*]
The values of ***print-level*** and ***print-length*** during description. Initially two and five.

extensions:*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.

extensions: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.

2.10. Modules

The CMU Common Lisp implementation of modules operates as described below in addition to conforming to the Common Lisp manual.

provide *module-name* [*Function*]
When a module is provided, *module-name* is added to ***modules*** indicating that it has been loaded. *Module-name* may be either a case-sensitive string or a symbol; if it is a symbol, its print name is downcased and used.

require *module-name* `&optional` *pathname* [*Function*]
When a module is required, it is loaded if it has not been already. *Module-name* may be either a case-sensitive string or a symbol; if it is a symbol, its print name is downcased and used. Pathname, if supplied, is a single pathname or list of pathnames to be loaded if the module needs to be loaded. If pathname is not supplied, then a list of files are looked for that were registered by a **extensions:defmodule** form. If the module has not been defined, then a file will be loaded whose name is formed by merging "modules:" and *module-name* (downcased if it is a symbol).

The following variable and macro are extensions to the Common Lisp module specification.

`extensions:*require-verbose*` *[Variable]*

While loading any files as a result of **`require`**, **`*load-verbose*`** is bound to **`*require-verbose*`** which defaults to `nil`.

`extensions:defmodule` *module-name* `&rest` *files* *[Macro]*

This defines a module by registering the files that need to be loaded when the module is required. *Module-name* may be either a case-sensitive string or a symbol; if it is a symbol, its print name is downcased and used.

2.11. Unix Interrupts

CMU Common Lisp allows access to all the Unix signals that can be generated under Mach. It should be noted that if this capability is abused, it is possible to completely destroy the running Lisp. The following macros and functions allow access to the Unix interrupt system. The signal names as specified in section 2 of the *Unix Programmer's Manual* are exported from the Mach package.

`system:with-enabled-interrupts` *specs,* `&rest` *body* *[Macro]*

The macro with-enabled-interrupts should be called with a list of signal specifications *specs*. Each element of *specs* should be a list of two or three elements: the first should be the Unix signal for which a handler should be established, the second should be a function to be called when the signal is received, and the third should be an optional character used to generate the signal from the keyboard. This last item is only useful for the SIGINT, SIGQUIT, and SIGTSTP signals. One or more signal handlers can be established in this way. With-enabled-interrupts establishes the correct signal handlers and then executes the forms in *body*. The forms are executed in an unwind-protect so that the state of the signal handlers will be restored to what it was before the with-enabled-interrupts was entered. A signal handler function specified as NIL will set the Unix signal handler to the default which is normally either to ignore the signal or to cause a core dump depending on the particular signal.

`system:without-interrupts` `&rest` *body* *[Macro]*

It is sometimes necessary to execute a piece a code that can not be interrupted. The macro without-interrupts executes the forms in *body* with interrupts disabled. Note that the Unix interrupts are not actually disabled, rather they are queued until after *body* has finished executing.

`system:with-interrupts` `&rest` *body* *[Macro]*

When an interrupt handler is called, interrupts are disabled, as if it is wrapped in without-interrupts. The macro with-interrupts can be used to enable interrupts while the forms in *body* are evaluated. This is useful if *body* is going to enter a break loop or do some long computation that doesn't need interrupts disabled.

`system:without-hemlock` `&rest` *body* *[Macro]*

For some interrupts, such as SIGTSTP (suspend the Lisp process and return to the Unix shell) it is necessary to leave Hemlock and then return to it. This macro executes the forms in *body* after exiting Hemlock. When *body* has been executed, control is returned to Hemlock.

`system:enable-interrupt` *signal function* `&optional` *character* *[Function]*

Enable-interrupt establishes *function* as the handler for *signal*. The optional *character* can be specified for the SIGINT, SIGQUIT, and SIGTSTP signals and causes that character to generate the appropriate signal from the keyboard. Unless you want to establish a global signal handler, you should use the macro with-enabled-interrupts to temporarily establish a signal handler. Enable-interrupt returns the old function associated with the signal and when *character* is specified for SIGINT, SIGQUIT, or SIGTSTP, the old character code.

system:ignore-interrupt *signal* [*Function*]

Ignore-interrupt sets the Unix signal mechanism to ignore *signal* which means that the Lisp process will never see the signal. Ignore-interrupt returns the old function associated with the signal or NIL if none is currently defined.

system:default-interrupt *signal* [*Function*]

Default-interrupt can be used to tell the Unix signal mechanism to perform the default action for *signal*. For details on what the default action for a signal is, see section 2 of the *Unix Programmer's Manual*. In general, it is likely to ignore the signal or to cause a core dump.

2.11.1. Default Interrupt Handlers for Lisp

CMU Common Lisp has several interrupt handlers defined when it starts up, as follows:

SIGINT	causes Lisp to enter a break loop. This puts you into the debugger which allows you to look at the current state of the computation. If you proceed from the break loop, the computation will proceed from where it was interrupted.
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.
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.
SIGILL, SIGBUS, and SIGSEGV	cause Lisp to signal an error.
SIGMSG	is a Mach specific signal that is generated when an IPC message is received. Most of the time this signal is ignored. However, when Lisp calls the function **server** when waiting for one of several things to happen, this signal is enabled and is used to return control to server when a message is received.
SIGEMSG	is another Mach specific signal that is generated when an IPC emergency message is received. The default action for Lisp is to immediately service the emergency message and any others that are pending.

The SIGINT, SIGQUIT, and SIGTSTP signals can be generated from the keyboard. The characters used to generate these interrupts are the same as in the shell. Generally, these are control-C for SIGINT, control-\ for SIGQUIT, and control-Z for SIGTSTP. Depending on what commands are in your .login or .cshrc files, the characters used to generate these interrupts may be different. When in the Lisp read-eval-print loop that you get by just running Lisp, these interrupts can be generated by typing the appropriate character. To generate one of these interrupts from the keyboard while running Hemlock depends on how Hemlock is run, as follows:

Console	On the bare console, SIGINT, SIGQUIT, and SIGTSTP are generated by typing F1, F2, and F3, respectively. These keys also work while not in Hemlock.
Under X	When running under the X window manager, SIGINT, SIGQUIT, and SIGTSTP are generated by typing the appropriate control character in the top-level Lisp window.
Terminal	When accessing Lisp from a normal terminal (either by telnet or terminal emulation mode under X), control-\ can be used to generate the SIGINT signal. The other interrupts can not be signalled directly while in Hemlock, but once in the debugger, they can be signalled by typing the appropriate character.

When a signal is generated, 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.

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.

2.11.2. Examples of Signal Handlers

The following code is the signal handler used by the Lisp system for the SIGINT signal.

```
(defun ih-sigint (signal code scp)
  (declare (ignore signal code scp))
  (without-hemlock
   (with-interrupts
    (break "Software Interrupt" t))))
```

The without-hemlock form is used to make sure that Hemlock is exited before a break loop is entered. The with-interrupts form is used to enable interrupts because the user may want to generate an interrupt while in the break loop. Finally, break is called to enter a break loop, so the user can look at the current state of the computation. If the user proceeds from the break loop, the computation will be restarted from where it was interrupted.

The following function is the Lisp signal handler for the SIGTSTP signal which suspends a process and returns to the Unix shell.

```
(defun ih-sigtstp (signal code scp)
  (declare (ignore signal code scp))
  (without-hemlock
   (reset-keyboard 0)
   (mach:unix-kill (mach:unix-getpid) mach:sigstop)))
```

Lisp uses this interrupt handler to catch the SIGTSTP signal because it is necessary to get out of Hemlock in a clean way before returning to the shell.

To set up these interrupt handlers, the following is recommended:

```
(with-enabled-interrupts ((mach:SIGINT #'ih-sigint 2)
                          (mach:SIGTSTP #'ih-sigtstp 26))
  <user code to execute with the above signal handlers enabled.>
)
```

2.12. 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.

extensions:save-lisp *file* **&key :purify :root-structures :init-function :load-init-file :print-herald :process-command-line** [*Function*]

The **save-lisp** function saves the state of the currently running lisp core image in *file*. **Save-lisp** is exported from the extensions package. The keyword arguments have the following meaning:

:purify If non-NIL (the default), the core image is purified before it is saved. This means moving accessible Lisp objects from dynamic space into read-only and static space. This reduces the amount of work the garbage collector must do when the resulting core image is being run. Also, if more than one Lisp is running on the same machine, this maximizes the amount of memory that can be shared between the two processes. Objects in read-only and static space can never be reclaimed, even if all pointers to them are dropped.

:root-structures This should be a list of the main entry points for the resulting core image. The

purification process tries to localize symbols, functions, etc., in the core image so that paging performance is improved. The default value is NIL which means that Lisp objects will still be localized but probably not as optimally as they could be. This argument has no meaning if *:purify* is NIL.

:init-function This is a function which is called when the saved core is resumed. The default function simply aborts to the top-level read-eval-print loop. If the function returns, it will be the value of **save-lisp**.

:load-init-file If non-NIL, then load an init file; either the one specified on the command line or init.fasl or, if init.fasl does not exist, init.lisp from the user's home directory. If the init file is found, it is loaded into the resumed core file before the read-eval-print loop is entered.

:print-herald If non-NIL, then print out the standard Lisp herald when starting.

:process-command-line
If non-NIL, processes the command line switches and performs the appropriate actions.

To resume a saved file, type:

```
lisp -c file
```

2.13. 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>) '(directory$_1$ directory$_2$ ...))

Where <name> specifies the search list and must be a string (case insensitive) terminated by a colon (:), and directory$_i$ are strings that specify Unix directories (case sensitive). 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.lisp to refer to the file /usr/lisp/code/eval.lisp.

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 as described above. 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/").

2.14. Running Programs from Lisp

It is possible to run programs from Lisp by using the following function.

extensions:run-program *program* **&optional** *args* **&key** **:env :wait :input** [*Function*]
:if-input-does-not-exist
:output ...

Run-program allows lisp to start up a child process and run the specified *program*. *Program* should be a string specifying the name of the file containing the program to run. *Args* (default NIL) should be a list of strings which are passed to *program* as normal Unix parameters. The following keyword arguments are defined:

:env	is a list of strings in the standard Unix environment format (i.e., "<variable>=<value>"). The default is to use the environment information passed to Lisp when Lisp was started. If *:env* is specified, it uses the value given and does not combine the environment passed to Lisp with the one specified.
:wait	If non-NIL (the default), wait until the child process terminates. If NIL, continue running Lisp while the child process runs.
:input	should be a string specifying the name of a file that contains input for the child process. This file will be opened on standard input. If value is NIL (default), then standard input is opened to the file "/dev/null". If value is T, the current standard input will be used. This may cause some confusion if the :wait argument is NIL, since two processes may be reading from the terminal at the same time.
:if-input-does-not-exist	specifies what to do if the input file does not exist. The valid values are: NIL (default) returns NIL from run-program without doing anything or **`:error`** signals an error.
:output	should be a string specifying the name of a file that will contain the output written to standard output by the child process. If value is NIL (default), all output is directed to "/dev/null". If value is T, standard output for the Lisp process is used. This may cause confusion, since two processes may be writing to the terminal.
:if-output-exists	specifies what to do if the output file already exists. The valid values are: NIL returns NIL from run-program without doing anything; **`:error`** (default) signals an error; **`:supersede`** overwrites the current file; and **`:append`** appends all output to the file.
:error	is similar to *:output*, except that the file is associated with standard error.
:if-error-exists	specifies what to do if the error output file already exists. It accepts the same values as *if-output-exists*.

All other file descriptors are closed in the child process before the program to run is invoked. If an error occurs, run-program returns NIL and possibly a second value specifying the Unix error that occurred. If run-program is successful, the values returned depend on the value of :wait. If :wait is non-NIL, then the values returned from the wait system call are returned. Otherwise, the process id of the child process is returned.

Chapter 3

Error System

By Bill Chiles
derived from Error System Proposal #8 by Kent M. Pitman

3.1. Introduction

The error system described in this chapter is close to becoming a Common Lisp standard, but it may change slightly before it is finally approved. It is built on a general *condition* system, with conditions being typed objects which are all subtypes of type **`condition`**. When an *exceptional situation* arises that is detected in a program, a condition object can be passed around the error system containing necessary information for other parts of the program or the user to take corrective action.

Conditions can be created explicitly with **`make-condition`** or implicitly by a function such as **`signal`** that can make a condition when given appropriate information. **`Signal`** is one means of announcing a condition to the error system when an exceptional situation occurs. When a condition is *signaled*, a *handler* is looked for. A handler is a function dynamically associated with a condition type, and it requires one argument, a condition, that the handler can inspect to determine which of the following actions to take:

- It may perform some non-local transfer of control using **`go`**, **`return`**, **`throw`**, or one of the error system's primitives built on these.
- It may signal another condition.
- It may invoke the interactive debugger.
- It may *decline* to handle the condition by returning. It is possible for a signaled condition to go unhandled. If the condition were signaled by **`error`** or **`break`**, and it is not handled by any handler, the interactive debugger will be called on the condition.

Some conditions are of type **`error`**. To *signal an error* is to signal a condition of type **`error`**, and the simplest way to do this is to call **`error`** on **`format`**-style arguments which results in the signaling of a **`simple-error`** condition.

Another, nearly distinct set of functionality provided by the error system is the interface to *proceed cases* and *proceed functions*. These are analogous to **`catch`** and **`throw`** since **`proceed-case`** forms set up dynamic points for non-local flow of control accessible by invoking proceed case objects. Unlike **`catch`**, **`proceed-case`** can setup more than one point of return for each use, and it allows the choice of naming each point while **`catch`** always needs a tag. Proceed cases that are not named are generally only accessible through the interactive debugger, but named ones may be invoked by handlers and other pieces of code.

3.2. General Error-signaling Functions

This sections describes functions used to signal various types of conditions. Most of these are described in the language standard, but they have changed upward compatibly to adhere to the new error system. Also, `debug` is described here since it is generally useful even though it does not signal any conditions.

These functions take a required *datum* argument and an *arguments* `rest` parameter for purposes of passing a condition either explicitly or implicitly. Unless otherwise specified, these are used as follows:

- If *datum* is a condition object, then that condition is used directly. In this case, it is an error for *arguments* to be supplied.
- If *datum* is a condition type, then the condition used is the result of `apply`'ing `make-condition` to *datum* and *arguments*.
- If *datum* is a string, then the condition used is the result of the following:

```
(make-condition 'simple-condition
                :format-string datum
                :format-arguments arguments)
```

`error` *datum* `&rest` *arguments* [*Function*]

Invokes the signal facility on a condition. If the condition is not handled, `debug` is called on it.

If *datum* is a string, then the condition used is the result of the following:

```
(make-condition 'simple-error
                :format-string datum
                :format-arguments arguments)
```

`cerror` *continue-string datum* `&rest` *arguments* [*Function*]

Invokes `error` on *datum* and *arguments* after setting up a proceed case named *proceed*. The report function for the proceed case is obtained from calling `format` on *continue-string* and *arguments*. Within the debugger and from any handlers bound at the time, the proceed function `proceed` will continue from this error because of the proceed case set up before calling `error`.

If *datum* is a condition object, then that condition is used directly. In this case, *arguments* will be used only with the *continue-string*, and it is not an error for it to be supplied.

If *datum* is a string, then the condition used is the result of the following:

```
(make-condition 'simple-error
                :format-string datum
                :format-arguments arguments)
```

Note that if *datum* is not a condition object, then *arguments* will be used both with *continue-string* and to make a condition with *datum*, so some care may need to be taken when supplying *arguments*. The format directive `~*`, which skips and backs up over arguments, will probably be useful in these cases.

If `cerror` returns, which it does when its proceed case is invoked, the value returned is the condition that was signaled.

`break &optional` *datum* `&rest` *arguments* [*Function*]

Invokes `debug` on *datum* and *arguments* after setting up a proceed case named *proceed*. Because of the proceed case, the proceed function `proceed` will cause `break` to return the condition used.

If *datum* is not supplied, it defaults to "Break".

`Break` could be written:

```
(defun break (&optional (datum "Break") &rest arguments)
  (proceed-case (apply #'debug datum arguments)
    (proceed (condition)
      :report "Return from BREAK."
      condition)))
```

warn *datum* **&rest** *arguments* *[Function]*

Invokes the signal facility on a condition. If the condition is not handled, then the condition is reported to ***error-output***. If ***break-on-warnings*** is non-nil, then after reporting the condition, a proceed case named *proceed* is set up, and the debugger is called on the condition. Because of the proceed case, the proceed function **proceed** will cause **warn** to return the condition used.

If *datum* is a string, then the condition used is the result of the following:

```
(make-condition 'simple-warning
                :format-string datum
                :format-arguments arguments)
```

Warn could be written:

```
(defun warn (datum &rest arguments)
  (let ((condition (apply #'signal datum arguments)))
    (format *error-output* "~&Warning: ~A~%" condition)
    (when *break-on-warnings*
      (proceed-case (debug condition)
        (proceed (condition)
          :report "Return from WARN."
          condition)))))
```

This example relies on **signal** to return the condition signaled when it is not handled; if it were handled, then the rest of the function would not be executed. For simplicity, this example is not written correctly since **signal** will signal a condition of type **simple-condition** when **datum** is a string, but **warn** must signal a **simple-warning** in this case.

signal *datum* **&rest** *arguments* *[Function]*

Tries to find a handler for a condition that is either handed to **signal** directly as *datum* or created implicitly from *datum* and *arguments.*

Handlers are located according to the following algorithm:

1. Check for any handler dynamically bound to the condition's type. These handlers are established with **condition-bind** or abstractions built on it.

2. Starting with the condition's type and searching hierarchically through its supertypes, check for a default handler. These handlers are established with **define-condition.**

If a handler is found, it is called. If the handler declines to handle the condition by returning (as opposed to performing some kind of special transfer of control), the search for a handler continues.

If no handler is found that actually handles the condition, **signal** returns the condition object. It is not necessary that a signaled condition be handled. Some conditions are not significant enough to break a system and may be signaled and ignored. Other conditions, those of type **serious-condition,** always end up in the debugger if no handler takes care of them.

debug &optional *datum* **&rest** *arguments* *[Function]*

Enters the debugger reporting a condition obtained either directly from *datum* or implicitly created from *datum* and *arguments.* Any proceed cases dynamically established at the time are also reported.

If *datum* is not supplied, then it defaults to the string **"Debug"**.

This function can never directly return. Return can occur only via special transfer of control, such as throwing to top level, proceeding the error, or using some debugger feature (see the Debugging Tools chapter).

3.3. Conditions and Handlers

This section describes the interface to defining condition types, instantiating them, and associating handlers with these types.

`define-condition` *name parent-type* {*keyword value*}* {*slot*}* [*Special form*]

Defines a new condition type with the given *name*, which is a subtype of the *parent-type*. The function **`typep`** operates on condition types. Except as otherwise noted, the arguments are not evaluated.

Valid *keyword/value* pairs:

`:conc-name` *symbol-or-string*
: As in **`defstruct`**, this sets up automatic prefixing of the names of slot accessors, defaulting to *name* followed by a hyphen.

`:report-function` *function*
: *Function* should be a suitable argument to the **`function`** special form. It designates a function of two arguments, a condition and a stream, that is called when **`*print-escape*`** is **`nil`** to print the condition to the stream

`:report` *form*
: This is a short form of **`:report-function`** provided for convenience in two common cases, when *form* is a constant string or a single call such as to **`format`**. If *form* is a string, using **`:report`** is similar to:

```
:report-function (lambda (condition stream)
                   (declare (ignore condition))
                   (write-string form stream))
```

Otherwise, this is similar to:

```
:report-function
(lambda (condition *standard-output*) form)
```

Actually, the form is executed in an environment with all slots of the condition accessible through variables of the same name, and the condition being printed will be the value of the variable **`condition`**. It is an error to assign any variables which have been set up in this way. The form should do output to **`*standard-output*`** since the condition may be reported at times other than when an error is being announced, and when an error is being announced, **`*standard-output*`** will be redirected to **`*error-output*`**. For example, the **`simple-error`** condition could be defined as follows:

```
(define-condition simple-error error
  :report
  (apply #'format t format-string format-arguments)
  format-string
  format-arguments)
```

`:handler-function` *function*
: *Function* should be a suitable argument to the **`function`** special form. It designates a function of one argument, a condition, to be the default handler for the condition type.

`:handle` *form*
: A form to be used as the body of a default handler for the condition type. The form is executed in an environment with all slots of the condition accessible through variables of the same name, and the condition being printed will be the value of the variable **`condition`**. It is an error to assign any variables which have been set up in this way.

It is an error to specify both `:report-function` and `:report` in the same `define-condition` form. If neither `:report-function` nor `:report` is specified, a reporting method for the condition type will be inherited from *parent-type*. The purpose of these printing methods is to provide the means to display conditions in a meaningful way for the user. When a condition is printed, and `*print-escape*` is `nil` (`format`'ing the condition with `~A` or `princ`'ing it), the report function for its type is invoked.

In addition to each *slot* description, the slots of *parent-type* are inherited. Each *slot* specification is either the name of a slot or a list. If it is a list, the first element is the name of the slot, and the second element is a form which will be executed by the constructor for the condition type to produce an initial value for the slot when one is not specified as an argument to the constructor. In either kind of *slot* specification, if no initial value form is provided, then `nil` is the default. If a slot name is specified which is the same as the name in some type of which this type is a subtype, only one shared slot is created in the condition object, but the specified initial value form overrides any which otherwise would have been inherited from a supertype.

Instances of the condition type can be created by calling `make-condition`, which knows how to invoke the constructor for this type. It will accept keywords with printnames of the specified slots and will initialize the corresponding slots in the returned condition.

Here are some examples of defining conditions. This form defines a condition type called `machine-error` which inherits from type `error`:

```
(define-condition machine-error error
  :report (format t "There is a problem with ~A." machine-name)
  machine-name)
```

The slot `machine-name` can be accessed with `machine-error-machine-name`, and `make-condition` will accept a `:machine-name` keyword when creating conditions of type `machine-error`.

This defines a condition subtype of `machine-error` to be used when machines are not available:

```
(define-condition machine-unavailable machine-error
  :report (format t "The machine ~A is not available."
                  machine-name))
```

The previous comments concerning `machine-error` apply to `machine-unavailable` conditions, and `machine-unavailable-machine-name` will also access the name of the machine having a problem.

This defines a still more specific condition type, a subtype of `machine-unavailable`, which provides a default for the `machine-name` slot:

```
(define-condition central-file-server-unavailable
                  machine-unavailable
  (machine-name "cfs.cs.cmu.edu"))
```

Since no `:report` or `:report-function` was specified, the report method for `machine-unavailable` conditions will be used when the error is printed with `*print-escape*` bound to nil.

Since all of these example conditions are subtypes of `error` (which is a subtype of `serious-condition`), and there are no default handlers specified, if one of these is signaled, and no handlers are currently bound to their types, the debugger will be invoked.

make-condition *type* **&rest** *slot-initializations* [*Function*]

Invokes the appropriate constructor function for *type*, passing along the given slot initializations to the constructor and returning an initialized condition. *Slot-initializations* is given as alternating keyword/value pairs. For example, if a condition type **peg/hole-mismatch** has slots named **peg-shape** and **hole-shape**, it could be instantiated in the following way:

```
(make-condition 'peg/hole-mismatch
                :peg-shape 'square :hole-shape 'circle)
```

condition-bind *bindings* {*form*}* [*Special form*]

Executes each *form* with the given handler bindings dynamically in effect. *Bindings* is a list of elements of the form **(types handler)**, where **handler** is a suitable argument for the **function** special form, and **types** is either a condition type or a list of types.

condition-case *form* {*case*}* [*Special form*]

Executes *form* after binding some handlers to condition types as specified by each *case*. Each *case* is of the following form:

```
(type ([var]) . body)
```

If a condition is signaled of the appropriate type and not handled by any intervening handlers, then control is transferred to a *case*, binding **var** (if specified) to the signaled condition. Any values returned by the body are returned by **condition-case**, and if no conditions are signaled, then any values returned by *form* are returned. *Type* is either a condition type or list of types.

If more than one case is provided, an implicit nesting occurs. For example,

```
(condition-case form
  (type1 (var1) form1)
  (type2 (var2) form2))
```

is equivalent to

```
(condition-case (condition-case form
                  (type1 (var1) form1))
  (type2 (var2) form2))
```

That is, earlier cases are found before later ones.

ignore-errors {*forms*}* [*Macro*]

Executes each *form* in a context which handles conditions of type **error** by returning **nil** from this form. If no **error** conditions are signaled, any values returned by the last *form* are returned.

Ignore-errors should not be used excessively since **error** conditions encompass too many kinds of errors. Keeping a system from dropping into the debugger does not necessarily correlate with reliability. It is better to only catch errors the program designer plans to handle and allow other unexpected errors to be discovered and interactively debugged.

Note that **error** conditions are a subtype of **serious-condition** conditions, and **ignore-errors** will not prevent the latter type from causing the debugger to be called.

Ignore-errors could be written:

```
(defmacro ignore-errors (&rest forms)
  `(condition-case (progn ,@forms)
     (error () nil)))
```

3.4. Proceed Cases and Defined Proceed Functions

Proceed cases and *proceed functions* are analogous to `catch` and `throw` since `proceed-case` forms set up dynamic points for non-local flow of control accessible by invoking proceed case objects. Unlike `catch`, `proceed-case` can setup more than one point of return for each use, and it allows the choice of naming each point while `catch` always needs a tag. Proceed cases that are not named are generally only accessible through the interactive debugger, but named ones may be invoked by handlers and other pieces of code.

A proceed case can have variables that will be bound during the execution of its body. These may be optional as in lambda-lists which allows proceed cases to be invoked safely when the variables are unknown to the caller. `Invoke-proceed-case`, which takes a proceed case name and other arguments, transfers control to a proceed case passing the arguments on to be bound to the variables specified by it. An example of this and the similarity of `catch` and `proceed-case` is shown by adding 3 to 1:

```
(proceed-case (invoke-proceed-case 'foo 3)
  (foo (x) (+ x 1)))
```

and

```
(+ (catch 'tag (throw 'tag 3)) 1)
```

A more practical example of the usefulness of `proceed-case` is:

```
(proceed-case (error 'unbound-variable :variable var)
  (nil ()
    :report (format t "Return the value :UNBOUND-VARIABLE.")
    :unbound-variable)
  (proceed (condition)
    :report (format t "Retry getting the value of ~S." var)
    (declare (ignore condition))
    (symbol-value var))
  (use-value (value)
    :report (format t "Specify a value of ~S to use this time." var)
    value)
  (store-value (value)
    :report (format t "Specify a value of ~S to store and use." var)
    (set var value)))
```

Note that proceed cases have report functions which are similar to condition report functions. This form establishes one unnamed proceed case, `nil`, and three named ones, `proceed`, `use-value`, and `store-value`. With `condition-bind`, a handler could be bound to conditions of type `unbound-variable` to arbitrarily provide `t` as a value:

```
(condition-bind ((unbound-variable
                  #'(lambda (condition)
                      (if (find-proceed-case 'use-value)
                          (invoke-proceed-case 'use-value t)))))
  body)
```

Note here that, unlike `catch` and `throw`, the existence of a certain named proceed case can be ensured before trying to transfer control to it. Using `define-proceed-function`, this idea (that is, testing for the proceed case before invoking it) can be encapsulated. Standard proceed functions are defined such as `proceed`, `use-value`, and `store-value` for convenience in writing handlers. The above `condition-bind` example could have been written as follows:

```
(condition-bind ((unbound-variable
                  #'(lambda (condition) (use-value t))))
  body)
```

The prime significance of having a named proceed case and correspondingly defined proceed function is that the proceed case can be specified with all its variables required, and `invoke-proceed-case` can use information from the `define-proceed-function` form to do argument defaulting. This allows standard proceed cases to be typed in quickly, while all the argument defaulting and user interaction information is entered once in the **define-proceed-function** form.

proceed-case *form* {*clause*}* [*Special form*]

Executes *form*, returning any values *form* returns, in a dynamic environment with each *clause* established as a proceed case. If a condition is signaled while *form* is executing, then control may be specially transferred to one of these proceed cases via **invoke-proceed-case**, a defined proceed function, or the debugger if the user ends up there.

Each *clause* has the following form:

```
(case-name arglist {keyword value}* forms)
```

The *case-name* is a symbol naming this proceed case. If the symbol is **nil**, it is considered that the proceed case is not named, and it cannot be found using **find-proceed-case**. More than one clause may use the same name, with the earlier uses being more recently bound than the later; that is, if two clauses use the name **proceed**, and the proceed function **proceed** is invoked, control will be transferred to the first such clause. All proceed cases are accessible via the combination of **compute-proceed-cases** and **invoke-proceed-case**.

The *arglist* is a list of variables to be bound during the execution of the *forms*, and optionals are allowed as in lambda-lists. These arguments are used to pass any necessary data from a condition handler to the proceed case clause. If a named proceed case corresponds to a defined proceed function, then the variables in *arglist* may be required while the proceed function's arguments are optional, and calling **invoke-proceed-case** directly or implicitly by calling the proceed function will resolve the optional arguments. This is the main convenience of named proceed cases; that is, a proceed case, requiring its variables, can be quickly written using a name corresponding to a defined proceed function, and the proceed function will know how to interact with the user to provide these values for the proceed case.

Valid *keyword/value* pairs:

:report-function *function*
: *Function* should be a suitable argument to the **function** special form. It designates a function of two arguments, a proceed case and a stream, that is called when ***print-escape*** is **nil** to report a summary to the stream of the action the proceed case will take.

:report *form*
: This is a short form of **:report-function** provided for convenience in two common cases, when *form* is a constant string or a single call such as to **format**. If *form* is a string, using **:report** is the same as:

```
:report-function (lambda (stream)
                   (write-string form stream))
```

Otherwise, this is the same as:

```
:report-function (lambda (*standard-output*) form)
```

It is an error to specify both **:report-function** and **:report** for the same proceed case. A default report method may be specified for a named proceed case with **define-proceed-function**, which is ignored if a report method is specified within the proceed case. If a report method is not specified in a named proceed case, and there is no default, then the name is printed. It is an error for an unnamed proceed case to not provide report information.

proceed-case-name *proceed-case* [*Function*]

Returns the name of *proceed-case*, or **nil** if it is not named.

compute-proceed-cases *[Function]*

Uses the dynamic state of the program to compute and return a list of proceed cases, ordered from more recently established to earlier ones.

Each proceed case represents a point to which control may be transferred via **invoke-proceed-case**. Proceed cases can be reported by printing them with ***print-escape*** bound to **nil**, such as with **princ** or the **format** directive **~A**, and **proceed-case-name** will return the name of each. **Find-proceed-case** will return the closest established proceed case with a given name in the dynamic environment, while **compute-proceed-cases** returns every proceed case, even those with the same name as more recently established, shadowing ones.

Implementations are permitted, but not required, to return non-**eq** lists from successive calls to **compute-proceed-cases** while in the same dynamic environment, and it is an error to modify the list that is returned by **compute-proceed-cases**.

find-proceed-case *name* *[Function]*

Searches for the most recently established proceed case with *name* in the current dynamic contour.

Name may be a proceed case object. If it is currently valid, then it is returned, otherwise nil.

It is an error for **nil** to be given as a *name*. **Compute-proceed-cases** allows access to unnamed proceed cases.

invoke-proceed-case *proceed-case* **&rest** *values* *[Function]*

Transfers control to *proceed-case*, passing *values* to it. *Proceed-case* must be a proceed case or the name of one that is valid in the current dynamic context. If *proceed-case* is not a valid argument, an error will be signaled.

If the proceed case is a named proceed case, and there is a correspondingly named proceed function, argument defaulting will be done as specified for the proceed function.

This operation is used primarily as a sub-primitive for implementing named proceed functions but may be useful when writing certain kinds of portable debugging tools.

define-proceed-function *name* {*keyword value*}* {*variable*}* *[Special form]*

Defines a function called *name* encapsulating the idea of

```
(if (find-proceed-case 'name)
    (invoke-proceed-case 'name args))
```

and providing optional argument resolution for proceed cases named *name*. This latter feature is important because a proceed case, requiring all its variables, can be succinctly and quickly written using a name corresponding to a proceed function, and the proceed function will know how to interact with the user to provide the values for the proceed case. The defined function either transfers control to a proceed case or returns **nil**.

Each *variable* is implicitly optional. As with lambda-lists, these can be specified as a symbol or list, whose first element is the name of the variable, whose optional second element is an initialization form, and whose optional third element is a supplied-p variable. If an initialization form is not given, it defaults to **nil**. Unlike normal functions, the supplied-p variables are only useful for initializing variables that follow, and these cannot be passed on to proceed cases.

Valid *keyword/value* pairs:

:report-function *function*

Function should be a suitable argument to the **function** special form. It designates a function of two arguments, a proceed case and a stream, that is called when `*print-escape*` is `nil` to report a summary to the stream of the action to be taken by a proceed case with the same name as the proceed function.

`:report` *form* This is a short form of `:report-function` provided for convenience in two common cases, when *form* is a constant string or a single call such as to `format`. If *form* is a string, using `:report` is the same as:

```
:report-function
(lambda (stream)
  (write-string form stream))
```

Otherwise, this is the same as:

```
:report-function
(lambda (*standard-output*) form)
```

It is an error to specify both `:report-function` and `:report`. This default reporting method is ignored if a report method is specified within a proceed case with the same name.

For example, let's take the following `proceed-case` form:

```
(let ((message (get-message (ed temp-file))))
  (loop
    (when
      (proceed-case (post-message message)
        (save-message (message file)
          (write-message message file)
          t)
        (edit-message ()
          (setf message
                (get-message (ed temp-file)))
          nil)
        (retry-posting (msg wait)
          (sleep wait)
          (setf message msg)
          nil)
        (abort (condition)
          :report "Punt message."
          t))
      (return))))
```

We want to loop until we successfully do something with the message bound by the `let`. After setting up four cases (`save-message`, `edit-message`, `retry-posting`, and `abort`) as possible means for proceeding any errors that occur dynamically within this context, we try to post the message.

To support the terse cases in the above example, we define some proceed functions that know about proceed cases named as above:

```
(define-proceed-function save-message
  :report "Save the message in a file."
  (message
   (file (prompt-for-file
          :prompt "File to save message in: "
          :default "bb.txt"))))
```

A proceed case named `save-message` will be expected to take two arguments. The first, `message`, defaults to nil, and the second, `file`, defaults to the result of prompting for a file.

```
(define-proceed-function edit-message
  :report "Edit the message and retry posting.")
```

A proceed case named `edit-message` will take no arguments, so calling this function will simply transfer control to the proceed case.

```
(define-proceed-function retry-posting
  :report "Try to post again after waiting."
  (message
   (wait (prompt-for-integer
          :prompt "Seconds to wait before posting: "
          :default 0))))
```

A proceed case named `retry-posting` will be expected to take two arguments. The first, `message`, defaults to nil, and the second, `wait`, defaults to the result of prompting for an integer.

`Abort` is predefined by the system, but we have specified a more specific report message than `"Abort."`

Now handlers could be defined that call these proceed functions. For example, the function `post-message` could automatically try to repost a message if the post error was a subtype of `net-timeout`:

```
...
(condition-bind
    ((net-timeout #'(lambda (condition)
                      (retry-post (post-error-message
                                   condition)))))
  (sub-post-primitive message))
...
```

If the posting primitive signals a condition that is a subtype of `net-timeout`, then the `lambda` will be invoked as a handler. The handler then tries to call the proceed function `retry-post`, providing the first argument as the message in the condition object. The second argument is defaulted as described when `retry-post` was defined.

Note that there is no reason for the body of the handler to look like this:

```
(if (find-proceed-case 'retry-post)
    (invoke-proceed-case
     'retry-post
     (post-error-message condition)))
```

The proceed function `retry-post` will return `nil` if no such proceed case exists. Furthermore, if there were no such established proceed cases, the handler would simply return indicating that it declined to handle the condition.

catch-abort *print-form* {*form*}* [*Macro*]

Sets up a **proceed-case** for the proceed function **abort**, using *print-form* as a value for the keyword **:report**.

If **abort** is not called while executing each *form*, all values returned by the last *form* are returned. If **abort** transfers control to the proceed case in **catch-abort**, then the values **nil** and the condition given to **abort** (possibly **nil**) are returned.

There must be at least one form in *forms*.

`Catch-abort` could be written:

```
(defmacro catch-abort (print-form form1
                       &rest more-forms)
  `(proceed-case (progn form1 ,@more-forms)
     (abort (condition)
       :report ,print-form
       (values nil condition))))
```

`proceed &optional` *condition* [*Function*]

A proceed function with default reporting:

`"Proceed with no special action."`

Condition defaults to `nil` when not supplied.

`abort &optional` *condition* [*Function*]

A proceed function with default reporting:

`"Abort."`

Condition defaults to `nil` when not supplied.

`use-value &optional` *value* [*Function*]

A proceed function with default reporting:

`"Specify a value to use this time."`

When not supplied, *value* is interactively prompted for.

`store-value &optional` *value* [*Function*]

A proceed function with default reporting:

```
"Specify a value to store permanently and use this
time."
```

When not supplied, *value* is interactively prompted for.

3.5. Predefined Conditions

This section describes all the predefined conditions in the system. A much more sophisticated initial set of conditions is proposed, but currently CMU Common Lisp does not signal these.

The definition mechanism for condition types only provides for hierarchical inheritance, but implementations with non-hierarchical type systems are allowed to use such systems as long as all subtype relationships specified are maintained. For example, it follows from the subtype descriptions in this section that in all implementations:

```
(typep c 'simple-error) implies (typep c 'error)
```

However, one cannot assume:

```
(typep c 'simple-error) implies (not (typep c 'simple-condition))
```

Portable code should not assume that the subtype relationships of conditions are mutually exclusive.

Following is a description of each predefined condition type:

`condition` All types of conditions must inherit from this type.

`warning` All types of warnings should inherit from this type. This is a subtype of `condition`.

`serious-condition`
Any condition, whether `error` or non-`error`, that should enter the debugger when signaled and not handled should inherit from this type. This is a subtype of `condition`.

`error` All types of errors should inherit from this type. This is a subtype of `serious-condition`. Condition objects of this type have a `function-name` slot in addition to any inherited.

`simple-condition`
When `signal`, `break`, and `debug` are given a `format`-style string and arguments, a condition of this type is created. This is a subtype of `condition`. Condition objects of this type have `format-string` and `format-argument` slots in addition to any inherited.

`simple-warning`
When `warn` is given a `format`-style string and arguments, a condition of this type is created. This is a subtype of `warning`. Condition objects of this type have `format-string` and

`format-argument` slots in addition to any inherited.

`simple-error`	When `error` and `cerror` are given a `format`-style string and arguments, a condition of this type is created. This is a subtype of `error`. Condition objects of this type have `format-string` and `format-argument` slots in addition to any inherited.

Chapter 4

Debugging Tools

By Jim Large, Steve Handerson, and Bill Chiles

4.1. 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. Currently, compiling a traced function compiles the encapsulation function (see below) and not the function being traced. Compiling a traced function from a Hemlock buffer changes the original definition of the function, i.e., it does the correct thing.

`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 function

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

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.

`extensions:*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.

`extensions:*trace-print-level*` [*Variable*]
`extensions:*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`**.

`extensions:*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.

4.1.1. 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. Compiling a function that has been encapsulated compiles the encapsulation function and not the original one. Compiling an encapsulated function from a Hemlock buffer changes the original definition of the function, i.e., it does the correct thing.

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.

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

extensions: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 be bound for the evaluation of *body*:

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

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

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

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

Encapsulate always returns *symbol*.

extensions: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.

extensions:encapsulated-p *symbol type* [*Function*]

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

4.2. The 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 **evaled**. 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***. Because of this, the stepper can not be used in Hemlock eval mode. When connected to a slave Lisp, the stepper can be used from Hemlock.

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.

extensions:*step-print-level* [*Variable*]
extensions:*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 4 and 5, respectively.

extensions:*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 40.

4.3. The Debugger

The debugger is an interactive command loop that allows a user to examine the function call stack. Whenever a *serious-condition* condition is signaled, and it is not handled, the debugger is invoked. Whenever **error** is called, and the condition it signals is not handled, the debugger is invoked. The debugger never directly returns, but commands are provided for proceeding errors, throwing to top level, and returning values from arbitrary frames.

Most commands refer to the current stack frame, though some take an optional argument that specifies a frame on which to operate. A number is assigned to each frame, starting with zero at the top, and the debugger's prompt includes the number of the current frame. Some debugger commands are symbolic (that is, the name of a symbol entered is interpreted as a command), and others are handled exactly as if they were valid forms one would type to the top level read-eval-print loop. There are two notable consequences of this: a user has a full Lisp read-eval-print loop at his disposal, but if he wants to evaluate certain symbols (those that are interpreted as debugger commands), he must enter `(eval '<symbol>)`. Except for processing the symbolic commands, the command loop maintains `*`, `+`, `/`, `-`, and friends.

The debugger can not be used in Hemlock's eval mode. When connected to a slave Lisp, the debugger can be used from within Hemlock.

See the Error chapter for a description of the **debug** function.

4.3.1. Frame Changing Commands

These commands move to a new stack frame and print the name of the function and the values of its arguments in the style of a Lisp function call. The printing is controlled by ***debug-print-length*** and ***debug-print-level***. A frame is *visible* if it has not been hidden by **debug:hide** (described below).

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 [*n*]	Move to a given frame, visible or not. Prompts for the number if not supplied.
s [*function-name* [*n*]]	Search down the stack for function. Prompts for the function name if not supplied. Searches an optional number of times, but doesn't prompt for this number; enter it following the function.
r [*function-name* [*n*]]	Search up the stack for function. Prompts for the function name if not supplied. Searches an optional number of times, but doesn't prompt for this number; enter it following the function.

4.3.2. Exiting Commands

These commands get you out of the debugger.

q	Throw to top level.
proceed [*n*]	Invokes the *n*th proceed case as displayed by the `error` command. If *n* is not specified, the available proceed cases are reported.
go	Calls `proceed` on the condition given to `debug`. If there is no proceed case named *proceed*, then an error is signaled.
abort	Effectively calls `abort` on the condition given to `debug`. This is useful for popping debug command loop levels and aborting to top level, as the case may be.
(debug:debug-return *expression* [*frame*]**)**	From the current or specified frame, return the result of evaluating expression. If multiple values are expected, then this function should be called for multiple values.

4.3.3. Information Commands

Most of these commands print information about the current frame or function, but a few show general information. A frame is *visible* if it has not been hidden by `debug:hide` (described below).

h	Displays a synopsis of debugger commands.
?	Calls `Describe` on the current function, displays number of local variables, and indicates whether the function is compiled or interpreted.
l	Lists the local variables in the current function. The values of the locals are printed, but their names are no longer available. An index is associated with each that can be used with `debug-local`.
p	Displays the current function call as it would be displayed by moving to this frame.
pp	Displays the current function call using `*print-level*` and `*print-length*` instead of `*debug-print-level*` and `*debug-print-length*`.
error	Causes the condition given to `debug` to report and and active proceed cases.
backtrace [*n*]	Displays all the visible frames from the current to the bottom. Only shows *n* frames if specified. The printing is controlled by `*debug-print-level*` and `*debug-print-length*`.
g	*Grinds* the current frame.
(debug:local *n* [*Frame*]**)**	Returns the value of the *n*th local variable in the current or specified frame.
(debug:argument *n* [*frame*]**)**	Returns the *n*th argument of the current or specified frame.
(debug:debug-function [*n*]**)**	Returns the function from the current or specified frame.
(debug:function-name [*n*]**)**	Returns the function name from the current or specified frame.
(debug:pc [*frame*]**)**	Returns the index of the instruction for the function in the current or specified frame. This is useful in conjunction with `disassemble`. The pc returned points to the instruction after the one that was fatal.

4.3.4. Other Commands

These commands deal with pushing command levels and hiding frames.

push	Recursively calls `debug` on the same condition object. This is useful in conjunction with the

other commands in this section, since aborting command loop levels restores previous hiding filters. Use the `abort` command to unwind command loops.

(debug:hide *option* [*what*])
: Makes the described stack frames invisible to the frame movement commands. The second argument may be a symbol or a list of symbols; the function returns the hidden members of the category. When *what* is not supplied, the current hidden items are returned. *option* is one of:

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

`:frame-type(s)`
: Specified frame types will not be visible. Currently, the only types of frames that can be talked about are `:catch` frames. These are hidden by default.

(debug:show *options what*)
: Cancels the effect of the corresponding **debug:hide**.

(debug:show-all) Make every frame visible, even those hidden by default.

(debug:hide-defaults)
: Make only the default things hidden.

4.3.5. Specials

These are the special variables that control the debugger action.

`extensions:*debug-print-level*` [*Variable*]
`extensions:*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 3 and 5, respectively.

`extensions:*debug-hidden-functions*` [*Variable*]

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

Chapter 5

The Compiler

5.1. 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 .fasl 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.

`extensions: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.

5.2. 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)).
```

5.3. Compiler Switches

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

`compiler::*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`.

`compiler::*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`.

`compiler::*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`.

`compiler::*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`.

`compiler::*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`.

`compiler::*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`.

`compiler::*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`.

`compiler::*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`.

`compiler::*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`.

`compiler::*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`.

`compiler::*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`.

5.4. 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:

- **`compiler::*peep-enable*`** is on unless **`cspeed`** is greater than **`speed`** and **`space`**.

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

Chapter 6

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.

6.1. 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 (*fasl*) 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.

6.2. 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.

6.3. 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 6.2).
- 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.

6.4. 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.

6.4.1. 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 6.2).
- 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.

6.4.2. 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.

6.4.3. 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. 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.

6.4.4. 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.

6.5. Simple Vs Complex Arrays

If an array is a **simple-string**, **simple-vector** or **simple-bit-vector**, more efficient code is generated 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.

6.6. 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.

6.7. 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.

6.8. 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.

6.9. 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.

Chapter 7

MACH Interface

By Rob Maclachlan and Skef Wholey

CMU Common Lisp attempts to make the full power of the underlying environment available to the Lisp programmer. This is done using combination of hand-coded interfaces, automatically generated MACH RPC stubs and foreign function calls to C libraries. Although the techniques differ, the style of interface is similar. This chapter provides an overview of the facilities available and general rules for using them, as well as describing specific features in detail. It is assumed that the reader has a working familiarity with Mach, Unix and X, as well as access to the standard system documentation.

7.1. Lisp Equivalents for C Routines

The MACH documentation describes the system interface in terms of C procedure headers. The corresponding Lisp function will have a somewhat different interface, since Lisp argument passing conventions and datatypes are different.

The main difference in the argument passing conventions is that Lisp does not support passing values by reference. In Lisp, all argument and results are passed by value. Interface functions take some fixed number of arguments and return some fixed number of values. A given "parameter" in the C specification will appear as an argument, return value, or both, depending on whether it is an In parameter, Out parameter, or In/Out parameter. The basic transformation one makes to come up with the Lisp equivalent of a C routine is to remove the Out parameters from the call, and treat them as extra return values. In/Out parameters appear both as arguments and return values. Since Out and In/Out parameters are only conventions in C, you must determine the usage from the documentation.

Thus, the C routine declared as

```
kern_return_t lookup(servport, portsname, portsid)
        port    servport;
        char    *portsname;
        int     *portsid;       /* out */
 {
  ...
  *portsid = <expression to compute portsid field>
  return(KERN_SUCCESS);
 }
```

has as its Lisp equivalent something like

```
(defun lookup (ServPort PortsName)
  ...
  (values
   success
   <expression to compute portsid field>))
```

An extra twist that complicates this "translation" process but makes programming easier is this: when the routine returns a record value, the components of that record may be returned as multiple values. This eliminates the need to extract fields from **Alien** structures (see below) and frees the programmer from having to explicitly deallocate such structures.

So, the C routine declared as

```
void getevent(servport, event)
        port    servport;
        keyevent *event;        /* out */

 {
  ...
  keyevent->cmd = <expression to compute Cmd field>
  keyevent->ch = <expression to compute Ch field>
  keyevent->region = <expression to compute Region field>
  keyevent->y = <expression to compute Y field>
  keyevent->x = <expression to compute X field>
  ...
 }
```

would be written like this in Lisp:

```
(defun getevent (servport)
  ...
  (values
   <expression to compute Cmd field>
   <expression to compute Ch field>
   <expression to compute Region field>
   <expression to compute Y field>
   <expression to compute X field>))
```

Fortunately, CMU Common Lisp programmers rarely have to worry about the nuances of this translation process, since the names of the arguments and return values are documented in a way so that the **describe** function (and the **Hemlock Describe Function Call** command, invoked with **C-M-Shift-A**) will list this information. Since the names of arguments and return values are usually descriptive, the information that **describe** prints is usually all one needs to write a call to a Matchmaker-generated function. Most programmers use this on-line documentation nearly all of the time, and thereby avoid the need to handle bulky manuals and perform the translation from barbarous tongues.

7.2. Type Translations

Lisp data types have very different representations from those used by conventional languages such as C. Since the system interfaces are designed for conventional languages, Lisp must translate objects to and from the Lisp representations. Many simple objects have a direct translation: integers, characters, strings and floating point numbers are translated to the corresponding Lisp object. A number of types, however, are implemented differently in Lisp for reasons of clarity and efficiency.

Instances of enumerated types are expressed as keywords in Lisp. Thus, an instance of the enumerated type defined by

```
Type KeyHowWait = (KeyWaitDiffPos, KeyDontWait, KeyWaitEvent);
```

would be written in Lisp as a keyword: **`:keywaitdiffpos`**, **`:keydontwait`**, or **`:keywaitevent`**.

Records, arrays, and pointer types are implemented with the **`Alien`** facility (see page 57.) Access functions are defined for these types which convert fields of records, elements of arrays, or data referenced by pointers into Lisp objects (possibly another object to be referenced with another access function):

- A record of type *type* can be constructed with a function **`make-`***type*. A field named *field* of a record of type *type* may be accessed with a function *type-field*, and set with **`setf`** of that function.
- An array of type *type* can be constructed with a function **`make-`***type*; if the array type allows for a variable upper bound on indices, these bounds may be specified. Elements of such an array may be accessed with the function *type*-**`ref`**, and may be set with **`setf`** of that function.
- A pointer of type *type* to an object may be dereferenced with a function **`indirect-`***type*. To create an object and get a pointer of type *type* to that object, one can call the function **`make-`***type*. If the pointer type references an array of objects, indices may be provided as optional arguments to the indirect function, and if the array has a variable upper bound, it may be specified when calling the constructor function.

One should dispose of **`Alien`** objects created by constructor functions or returned from remote procedure calls when they are no longer of any use, freeing the virtual memory associated with that object. Since **`Aliens`** contain pointers to non-Lisp data, the garbage collector cannot do this itself. If the **`Alien`** was created using MACH memory allocation (e.g. **`vm_allocate`**), then the storage should be freed using **`dispose-alien`**. If the memory was obtained from a foreign function call to a routine that used **`malloc`**, then **`system:free`** should be used on the **`system:alien-sap`** of the **`Alien`**.

Note that in some cases an address is represented by a Lisp integer, and in other cases it is represented by a real pointer. Pointers are usually used when an object in the current address space is being referred to. The MACH virtual memory manipulation calls must use integers, since in principle the address could be in any process, and Lisp cannot abide random pointers. Because these types are represented differently in Lisp, one must explicitly coerce between these representations.

`system:alien-sap` *alien* [*Macro*]

The function **`alien-sap`** is used to generate a system area pointer (a virtual address that points into the section of Lisp's address space reserved for **`Alien`** objects) from an **`Alien`**.

NOTE: Usually a pointer from a system interface function is an **`Alien`**, but whenever a pointer is passed in, it must be passed as a system area pointer. This strange calling convention was adopted to eliminate the necessity of constructing an **`Alien`** value just for the purpose of passing a pointer. The programmer interface is simplified, since a simple function call can be made in most places without the need to declare a local variable, construct an **`Alien`** value, and deallocate that **`Alien`** value (or use **`alien-bind`** for those three things).

`system:sap-int` *sap* [*Macro*]
`system:int-sap` *int* [*Macro*]

The function **`sap-int`** is used to generate an integer corresponding to the system area pointer, suitable for passing to the kernel interfaces (which wants all addresses specified as integers). The function **`int-sap`** is used to do the opposite conversion.

7.3. Unix System Calls

You probably won't have much cause to use them, but all the Unix system calls are available. The Unix system call functions are in the "`mach`" package. The basic name is prefixed with "`unix-`" to prevent name conflicts. The associated constants usually don't have any prefix. To find out how to use a particular system call, try describing it. If that doesn't help, look at the source in syscall.lisp or consult your system maintainer.

The Unix system calls indicate an error by returning `nil` as the first value and the Unix error number as the second value. If the call succeeds, then the first value will always be non-null, often `t`.

`mach:get-unix-error-msg` *error* [*Function*]

Return a string describing the Unix error number *error*.

7.4. Making Sense of Return Codes

Whenever a remote procedure call returns a Mach error code (such as `kern_return_t`), it is usually prudent to check that code to see if the call was successful. To relieve the programmer of the hassle of testing this value himself, and to centralize the information about the meaning of non-success return codes, CMU Common Lisp provides a number of macros and functions.

`system:gr-error` *function gr* `&optional` *context* [*Function*]

Signals a Lisp error, printing a message indicating that the call to the specified *function* failed, with the return code *gr*. If supplied, the *context* string is printed after the *function* name and before the string associated with the *gr*. For example:

```
* (gr-error 'nukegarbage 3 "lost big")

Error in function GR-ERROR:
NUKEGARBAGE lost big, no space.
Proceed cases:
0: Return to Top-Level.
Debug  (type H for help)
(Signal #<Conditions:Simple-Error.5FDE0>)
0]
```

`system:gr-call` *function* `&rest` *args* [*Macro*]
`system:gr-call*` *function* `&rest` *args* [*Macro*]

These macros can be used to call a function and automatically check the GeneralReturn code and signal an appropriate error in case of non-successful return. `gr-call` returns `nil` if no error occurs, while `gr-call*` returns the second value of the function called.

```
* (gr-call mach:port_allocate *task-self*)
NIL
*
```

`system:gr-bind ({`*var*`}*) (`*function* `{`*arg*`}*) {`*form*`}*` [*Macro*]

This macro can be used much like `multiple-value-bind` to bind the *vars* to return values resulting from calling the *function* with the given *args*. The first return value is not bound to a variable, but is checked as a GeneralReturn code, as in `gr-call`.

```
* (gr-bind (port_list port_list_cnt)
           (mach:port_select *task-self*)
    (format t "The port count is ~S." port_list_cnt)
    port_list)
The port count is 0.
#<Alien value>
*
```

7.5. Packages

The functions and constants that make up each Matchmaker-generated interface usually reside in their own package, and the public symbols of that package are exported. Thus, one usually **uses** the package for an interface one wishes to use. A program that used the Mach kernel, the X window manager, and the Message Name server might begin with:

```
;;; -*- Package: Hack -*-
;;;
;;; A silly graphics hack.
;;; Written by Joe Schmoe.
;;;
(in-package "HACK" :use '("LISP" "X" "MACH" "MSGN"))
```

Note that all of the standard interfaces are built into the CMU Common Lisp core image, and one doesn't need to load any other files to use these facilities. Here is a list of the packages that hold the built-in interfaces.

MACH	Holds the MACH interface and the Unix system calls.
MSGN	Holds code for message name server calls.
X	Holds a foreign function call interface to the C Xlib. See the Xlib documentation for details. Note: multiple displays are not currently supported by Lisp. The system assumes that the most recently opened display is always current. If you open other displays, then programs (such as **Hemlock**) the user **server** will break.
TS, EVAL	Holds Matchmaker interfaces used to control Lisp client processes from Hemlock.

7.6. Useful Variables

The information passed to the process in its startup message is available in the values of global variables.

`system:*nameserverport*` *[Variable]*

Port to the message name server.

`system:*task-self*` *[Variable]*
`system:*task-data*` *[Variable]*
`system:*task-notify*` *[Variable]*

The initial ports for the Lisp process.

7.7. Reading the Command Line

The shell parses the command line with which Lisp is invoked, and passes a data structure containing the parsed information to Lisp. This information is then extracted from that data structure and put into a set of Lisp data structures.

extensions:*command-line-strings* [*Variable*]
extensions:*command-line-utility-name* [*Variable*]
extensions:*command-line-words* [*Variable*]
extensions:*command-line-switches* [*Variable*]

The value of ***command-line-words*** is a list of strings that make up the command line, one word per string. The first word on the command line, i.e. the name of the program invoked (usually "**lisp**") is stored in ***command-line-utility-name***. The value of ***command-line-switches*** is a list of **command-line-switch** structures, with a structure for each word on the command line starting with a hyphen. All the command line words between the program name and the first switch are stored ***command-line-words***.

The following functions may be used to examine **command-line-switch** structures.

extensions:cmd-switch-name *switch* [*Function*]

Returns the name of the switch, less the preceding hyphen and trailing equal sign (if any).

extensions:cmd-switch-value *switch* [*Function*]

Returns the value designated using an embedded equal sign, if any. If the switch has no equal sign, then this is null.

extensions:cmd-switch-words *switch* [*Function*]

Returns a list of the words between this switch and the next switch or the end of the command line.

7.8. Reading and Writing Virtual Memory Without Aliens

It is sometimes necessary to bypass the **Alien** type system and access virtual memory directly. The following functions are used to examine virtual memory:

system:sap-ref-8 *sap offset* [*Function*]

Returns the 8-bit byte at *offset* bytes from the *sap* as an integer in the range 0 to 255.

system:sap-ref-16 *sap offset* [*Function*]

Returns the 16-bit word at *offset* words beyond the *sap* as an integer in the range 0 to 65535.

system:sap-ref-32 *sap offset* [*Function*]

Returns the 32-bit dualword at *offset* (16-bit) words beyond the *sap* as a signed 32-bit integer.

Setf may be used with the above functions to deposit values into virtual memory.

7.9. The Software Interrupt System

There are default handlers for most of the Unix signals. The most interesting signal is the one that indicates an emergency message has arrived. Emergency message interrupts are enabled by default. When an emergency message arrives, the object-set mechanism is used to find a handler function (see page 53). It is as though **server** was called asynchronously.

system:add-port-death-handler *port function* [*Function*]
system:remove-port-death-handler *port function* [*Function*]

add-port-death-handler makes *function* a handler for port death on *port*. There may be any number of port death handlers for a port. When the port dies, all the handlers are with the port as an argument.

remove-port-death-handler undoes the effect of **add-port-death-handler**.

system:*port-receive-rights-handlers* [*Variable*]
system:*port-ownership-rights-handlers* [*Variable*]

These variables hold hashtables from ports to functions. When an ownership or receive rights message is received on a port, the port is looked up in the appropriate table. If there is a handler function, then it is called with the port as an argument.

system:*port-death-handlers* [*Variable*]
system:*pornography-of-death* [*Variable*]

The **dataport**, on which port death messages are sent, is associated with a function that consults another hash table, ***port-death-handlers***. If no associated function is found, then the port death handler quietly returns, unless ***pornography-of-death*** is **nil**, in which case a warning is printed. If a handler is found, it is called with the dead port.

Because interrupt handlers may do arbitrarily hairy things, one must be careful when writing non-reentrant code that might be called from an interrupt handler. The macro **without-interrupts** (**without-interrupts** (page 10)) may be used to execute forms with the interrupt system effectively turned off, should be used in such situations.

Chapter 8

Event Dispatching with SERVER

By Rob Maclachlan

It is common to have multiple activities simultaneously operating in the same Lisp process. Furthermore, Lisp programmers tend to expect a flexible development environment. It must be possible to load and modify application programs without requiring modifications to other running programs. CMU Common Lisp achieves this by having a central scheduling mechanism based on an event-driven, object-oriented paradigm.

An *event* is some interesting happening that should cause the Lisp process to wake up and do something. The two main kinds of events are MACH IPC messages and X events. It is also possible to wait for data on Unix file descriptors. This capability is somewhat different, and is described later.

MACH IPC and X events are conceptually fairly similar, so server handles them in much the same way. Both contain an *object capability* and an *operation code*. In a Mach IPC message the object capability is the remote port in the message header, and the operation code is the message ID. In an X event, the window ID is the object capability and the X event type is the operation code.

8.1. Object Sets

An *object set* is a collection of objects that have the same implementation for each operation. Externally the object is represented by the object capability and the operation is represented by the operation code. Within Lisp, the object is represented by an arbitrary Lisp object, and the implementation for the operation is represented by an arbitrary Lisp function. The object set mechanism maintains this translation from the external to the internal representation.

`system:make-object-set` *name* **`&optional`** *default-handler* [*Function*]

Makes a new object set. *name* is a string, which is used only for purposes of identifying the object when it is printed. *default-handler* is the function which is used as a handler when an undefined operation is done on an object in the set. Initially the object set has no objects and no defined operations.

`system:object-set-operation` *object-set operation-code* [*Function*]

Return the function which is the implementation of the operation corresponding to *operation-code* in *object-set*. When set with **`setf`**, changes the implementation. Usually this function is not called directly, since the object set operation is implicitly set by the **`serve-`***operation* functions in the X interface or Matchmaker generated interface.

system:add-xwindow-object *window object object-set* [*Function*]
system:add-port-object *port object object-set* [*Function*]

These functions add a new object to *object-set*. *Object* is an arbitrary Lisp object that is associated with the object capability *window* or *port*. *Window* is an X window ID, and *port* is a MACH IPC port. When an event happens, *object* is passed as the argument to the handler function.

8.2. The SERVER Function

The **server** function is the standard way for an application to wait for something to happen. For example, the **Hemlock** editor calls **server** when it wants input from X or an ASCII terminal. The idea behind **server** is that knows the appropriate action to take when any interesting event happens. If an application calls **server** when it is idle, then any other applications with pending events can run. This allows several applications to run "at the same time" without interference, even though there is only one thread of control.

system:server &optional *timeout* [*Function*]

Wait for an event to happen, and then dispatch to the correct handler function. If specified, *timeout* is the number of seconds to wait before timing out. A time out of zero seconds is legal and will cause server to poll to see if any events should be processed. Server returns T if at least one event has been serviced, and NIL otherwise. When using server and it returns T, it should be called repeatedly (with a timeout of 0) until it returns NIL.

If a MACH IPC message is received, then the **system:server-message** global **Alien** variable will contain the received message. Similarly, if an X event is received, then **system:server-event** will hold the event. See the MACH Interface chapter (page 45) for the details of using port objects.

If input is available on any designated file descriptor, then the appropriate handler function will be called. See ***file-input-handlers*** below.

Since events for many different applications may be arriving simultaneously, an application that is waiting for a specific event must loop calling server until the desired event happens. Since programs such as **Hemlock** call **server** to do input, applications such as Matchmaker servers usually don't need to call server at all; **Hemlock** will process the requests when it does into an input wait.

system:*file-input-handlers* [*Variable*]

This variable is an alist from Unix file descriptors to handler functions. If input is available on any of the file descriptors, then the corresponding function will be called with the file descriptor as its argument.

8.3. Using SERVER with Matchmaker Interfaces

We define the Object-Set to be a collection of objects (ports) with some set of operations (message ID's) with corresponding implementations (functions).

Matchmaker uses the Object-Set operations to implement servers. For each server interface *XXX*, a function of two arguments **serve-***XXX* is defined. The **serve-***XXX* function establishes the function which is its second argument as the implementation of the XXX operation in the object-set which is its first argument. The function is called with the *object* given to **add-port-object** as its first argument, and the input parameters as the remaining arguments. The return values from the function are used as the output parameters for the message (if any). **serve-***XXX* functions are also generated for each "server message" and asynchronous user interface.

In order to use a Lisp server, create an object set, define some operations on it using **serve-***XXX* functions, create an object for every port you want to receive on, and then call the **server** function to serve an RPC request.

In case it isn't obvious why things are done this way, consider that object sets allow there to be many servers in the same lisp which can function without knowing about each other. There can even be multiple different implementations of the same interface. This property is especially useful when handling emergency messages, since emergency message handling now uses the same mechanism.

8.4. Using SERVER with the X Interface

When an X event is available on the current display, then **server** uses the object associated with the window ID to find the handler function. Each X event type has a hand-generated **serve-***XXX* function similar to those generated for Matchmaker interfaces.

8.5. A SERVER Example

This sections presents a very simple example of the use of the server function. It defines a *xwindows* object set, sets up a handler for the X keypressed event, and makes calls to server to serve the keypressed events.

```
(in-package "SERVER-EXAMPLE" :use '("LISP" "EXTENSIONS" "SYSTEM" "X"))

(defvar *xwindows* (make-object-set "X Windows"))

(defun key-pressed (window-id modifiers code x y subwindow
                              time location-x location-y)
  "Key-pressed is called when a key pressed is generated by X.
  It accepts the window in which the event happened.  The modifier
  bits (if any), the code for the key, the x and y position within
  the window of the mouse at the time the key was pressed, the
  subwindow (if any), the time of the event, and the absolute x
  and y location of the mouse."
  (declare (ignore modifiers x y subwindow time location-x location-y))
  (format t "Key-pressed (Window = ~D) = ~D.~%" window-id code))

(serve-keypressed *xwindows* #'key-pressed)


(defun server-example ()
  "An example of using the server function."
  ;; Create an Xwindow.  Assume xopendisplay has already been called.
  (let ((window (xcreatewindow (rootwindow) 0 0 512 512
                               0 (blackpixmap) (whitepixmap))))
    ;; Wrap code in unwind-protect, so we clean up after ourselves.
    (unwind-protect
        (progn
          ;; map the window to the screen.
          (xmapwindow window)
          ;; enable keypressed events.
          (xselectinput window x:keypressed)
          ;; Add the window to the *xwindows* object set.
          (add-xwindow-object window window *xwindows*)
          ;; Make sure the window gets displayed.
          (xflush)
          ;; Call server for 100,000 events.
          (dotimes (i 100000) (server)))
      ;; Remove the window from the object set before destroying it.
      (remove-xwindow-object window)
      ;; Destroy the window.
      (xdestroywindow window)
      ;; Make sure X destroys window NOW.
      (xflush))))
```

Other X events could be handled by selecting the various event types and adding servers for the event type to X window object set.

Chapter 9

The Alien Facility

By Rob Maclachlan

9.1. What the Alien Facility Is

`Aliens` provide a mechanism in Lisp for manipulating objects which are foreign to the Lisp environment. `Aliens` are used in the foreign function calling interface, matchmaker interfaces to the Mach specific system calls and the name server, and to call Unix system calls. The `Alien` functions and macros described in this chapter allow Lisp objects to be converted from the Lisp representation to other representations as expected in C code or IPC messages and vice versa.

9.2. 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.

9.3. 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.

9.4. 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.

`system: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`**.

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

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

`system: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.

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

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

`system: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.

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

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

`system: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`.
`(system:enumeration` *name***`)`**	Access a value of the enumeration *name*. Enumerations are defined by the macro **`defenumeration`** (page 59).

string-char An eight-bit ASCII character.

simple-string The corresponding **Alien** type is **system:perq-string** which is a Perq Pascal string (a string whose first byte is a count of the remaining characters). A second **Alien** type **system:null-terminated-string** has been defined which allows passing and receiving C style strings.

system:port A Mach IPC port.

short-float long-float
There are two alien types one for **short-float** and one for **long-float**. The **long-float** type is used as is without any loss of precision. Deporting a **short-float** from Lisp causes the four lowest mantissa bits to be set to 0 in a 32 bit word. Importing a **short-float** causes the four lowest order mantissa bits to be lost. If you want accuracy, you must use the **long-float** format.

system: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, additional types are available:

(system: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.

(system: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.

system:defenumeration *name* {{*element*}$^+$ | {(*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)

(setf (alien-access (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))
```

9.5. 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.

system:alien-value *name* [*Special form*]

Return the value of the **Alien** variable *name*.

system: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.

system: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 58). *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.

9.6. 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.

system: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.

system: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*.

9.7. 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.

system: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 60). 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.

system: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

system:alien-indirect *alien size* [*Function*]

This function takes a word 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**.

system:long-words *n* [*Function*]
system:words *n* [*Function*]
system:bytes *n* [*Function*]
system: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.

9.8. Examples

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

```
struct foo {
    int a;
    struct foo *b[100];
};

struct foo f;
```

<==>

```
;;; This operator selects the A field from a Foo.  The type of the
;;; resulting Alien is (signed-byte 32), which is what a C int 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 thirty-two bits,
;;; or one  long-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 (long-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 long-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) (long-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 two long-words.
;;;
(defoperator (deref-foo foo) ((rfoo (ref foo)))
  `(alien-indirect (alien-value ,rfoo) (long-words 2)))

;;; 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 (long-words 2))
```

With this definition, the following C 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))
```

Chapter 10

Foreign Function Call Interface

By David B. McDonald

10.1. Introduction

The foreign function call interface allows a Lisp program to call functions written in other languages. The current implementation of the foreign function call interface assumes a C calling convention and thus routines written in any language that adheres to this convention may be called from Lisp. Several functions and macros are made available to load object files into the currently running Lisp, to define data structures to be passed to or received from foreign routines, and to define the interface to a foreign function.

The foreign function call interface relies heavily on the primitives provided by the alien facility. If you intend to use the full power of the foreign function call interface, you will need to become familiar with the facilities provided by aliens. See the previous chapter for details.

Lisp sets up various interrupt handling routines and other environment information when it first starts up and expects these to be in place at all times. The C functions called by Lisp should either not change the environment, especially the interrupt entry points, or should make sure that these entry points are restored when the C function returns to Lisp. If a C function makes changes without restoring things to the way they were when the C function was entered, there is no telling what will happen.

10.2. Loading Unix Object Files

There is a single function that loads in one or more Unix object files into the currently running Lisp.

`extensions:load-foreign` *files* `&optional` *libraries linker base-file env* [*Function*]

Load-foreign loads a list of Unix object files into the currently running Lisp. *Files* should be a simple-string specifying the name of a single Unix object file or a list of such strings. *Libraries* should be a list of simple-strings specifying libraries in a format that ld, the Unix linker, expects. The default value for *libraries* is '("-lc") (i.e., the standard C library). *Linker* should specify the Unix linker to use when linking the object files. The default is "/usr/cs/bin/ld". *Base-file* is the file to use for the initial symbol table information. The default is the Lisp start up code ("/usr/misc/.lisp/bin/lisp"). *Env* should be a list of simple strings in the format of Unix environment variables (i.e., "A=B", where A is an environment variable and B is its value).The default value for *env* is the environment information available at the time Lisp was invoked. Unless you are certain that you want to change this, you should just use the default.

Load-foreign runs a Unix linker (default "/usr/cs/bin/ld") on the files and libraries (in the order given to load-foreign) creating an absolute Unix object file. This object file is then loaded into a memory at the correct location. All the external symbols that define either routines or variables are placed in a hash table for use by the macros that define interfaces to foreign routines. Note that load-foreign must be run before the any references to foreign functions or variables are made.

10.3. Defining Foreign Data Types

There are several data types that are pre-defined and can be used directly for defining interfaces to routines. There are also facilities for defining more complicated data structures such as arrays, structures, and pointers.

The following table gives a list of the pre-defined data types and the corresponding Lisp data type provided by the foreign function interface:

C Data Type	**Lisp Data Type**
int or long	(signed-byte 32)
unsigned int or long	(unsigned-byte 32)
short	(signed-byte 16)
unsigned short	(unsigned-byte 16)
char	(signed-byte 8)
unsigned char	(unsigned-byte 8)
float	short-float
double	long-float
procedure pointer	system:c-procedure

If you need to know how many bits are being used to represent a particular data structure, you can use the following function.

`extensions:c-sizeof` *c-type* [*Function*]

C-sizeof accepts a C type specification and returns the number of bits needed to represent it. For example, (**`c-sizeof`** 'int) returns 32.

10.3.1. Defining New C Types

`extensions:def-c-type` *name spec* [*Macro*]

Def-C-Type defines the symbol *name* to be a C type as specified by *spec*. *Spec* can either be a previously defined C type, or an alien type such as (signed-byte 32) or (system:null-terminated-string 256). This mechanism provides a short hand for referring to a particular type in other definitions.

For example, int above is defined by the following call to def-c-type:

```
(def-c-type int (signed-byte 32))
```

10.3.2. Defining C Arrays

`extensions:def-c-array` *name element-type* **`&optional`** *size* [*Macro*]

Def-C-array defines a C array type with name *name*. *Element-type* specifies the type of each element of the array. The optional parameter *size* specifies the number of elements in the array.

Def-C-array creates the following functions and forms that can be used to manipulate a C array:

make-*name* This function is used to allocate an array. Note that def-c-array does not actually create any storage for the array. You must use this routine to do that. If the *size* parameter is specified in the call to def-c-array, then make-*name* accepts no arguments and returns an alien value of the appropriate size. Otherwise, it accepts one argument which should be the number of elements desired for this particular instantiation of the array. In either case, an alien value is returned and can be used to refer to the storage for the array.

name-ref
: This setfable form allows you to refer to a particular element of an array. It accepts two arguments an alien value such as returned by make-*name* and an index. It picks up the correct element out of the array and returns it as the value. You can use setf on this form to set an element of an array.

For example, it is possible define an array type, create and instance of it, and set the first element of the newly created instance with the following code:

```
(def-c-array arr int 10)
(setq x (make-arr))
(setf (alien-access (arr-ref x 0)) 10)
```

10.3.3. Defining C Records

extensions:def-c-record *name* {(*sname stype*)}* [*Macro*]

Def-c-record defines a C record. This macro actually defines two C types *Name* is the name of the record and **Name* is the name of the pointer to the record. This is useful for record structures that have pointers to themselves as one or more of the slots. Following the *name* of the record are a list of (*sname stype*) pairs. These are the name and the type of a slot, respectively. As with def-c-array, def-c-record does not allocate any storage to hold data. It just defines the type. It also defines the function make-*name* which can be used to create an instance of the record. This will allocate storage to hold the record and return an alien value that refers to that particular record. For each field in the record, a setfable operator (named *name-sname*) is created, so that it is possible to reference and set particular fields of a record.

As an example, the following C structure definition and lisp def-c-record define equivalent data structures:

```
struct c-struct {
        short x, y;
        char a, b;
        int z;
        c-struct *n;
};

(def-c-record c-struct
        (x short)
        (y short)
        (a char)
        (b char)
        (z int)
        (n *c-struct))
```

To create an instance of c-struct and assign values to fields, the following code could be used:

```
(setq cs (make-c-struct))

(setf (alien-access (c-struct-x cs)) 20)
(setf (alien-access (c-struct-a cs)) 5)
(setf (alien-access (c-struct-n cs)) (lisp::alien-value-sap cs))
```

10.3.4. Defining C Pointers

C allows one to have pointers to other C-types. This can be done using the def-c-pointer macro as follows:

`extensions:def-c-pointer` *name to* [*Macro*]

Def-c-pointer defines *name* to be a C type that is a pointer to the C type specified by to.

For example, it is possible to define a pointer to an int by the the following:

```
(def-c-pointer *int int)
```

10.4. Defining Variable Interfaces

It is sometimes necessary to be able to refer to a global C variable. The macro def-c-variable allows this.

`extensions:def-c-variable` *name type* [*Macro*]

Def-c-variable makes global C variables accessible from Lisp. *Name* should be a simple-string with the exact capitalization of the C variable to which you want to be able to refer (C is case sensitive and so must be the name provided). This macro creates a Lisp symbol with *name* (uppercased) whose value is an alien value with type *type* that can be used to access the global C variable.

For example, it is often necessary to read the global C variable errno to determine why a particular function call failed. It is possible to define errno and make it accessible from Lisp by the following:

```
(def-c-variable "errno" int)
```

Now it is possible to get the value of the C variable errno by doing the following:

```
(alien-access errno)
```

10.5. Defining Routine Interfaces

There is a single macro that defines the interface to a C function. Note that all the types that it uses must be defined before you define the interface, otherwise errors will occur.

`extensions:def-c-routine` *name rtype* `&rest` *spec* [*Macro*]

Def-c-routine defines a Lisp function that interfaces to a C routine. *Name* should be a simple string with the exact capitalization of the C function (since C is case sensitive) or a list of two elements. The first element should be a simple-string as above and the second should be a symbol which is used as the Lisp name of the function. If this second form is not used, a symbol with *name* uppercased is used as the name of the Lisp function.

Rtype is the type of the return value and should be one of the builtin C-types or a user defined one. The special type extensions:void can be used if the C routine returns no useful value as its standard return value. Currently, double floats can not be returned by C functions. If the function returns a pointer and the result coming back is C NULL (0), then the function will return NIL. Also, if the result is a C String, then a Lisp string is returned instead of the alien value pointing to the C string.

Spec is bound to a list of the rest of the forms in the call to def-c-routine. Each element of this list should have the following form:

(*aname atype* `[`*amode*`] {options}*)`

Where *aname* should be a symbol and is used as the name of the argument. *Atype* should be a symbol that is associated with a C type. If you are passing floating point numbers to a C routine, you should declare the type of the parameter as a long-float or double. This is because C passes all floating point parameters as double floats. The routine may be called with any type of number, since it will be coerced to a long-float before being passed as on to C. *Options* is currently ignored. *Amode* should be one of the following:

:in	This specifies that the argument is passed by value and is the default. No value for this argument is returned by the Lisp function when this mode is used.
:out	The type of the argument must be a pointer to a fixed sized object (such as a record or fixed size array). An object of the correct size is allocated and passed to the C routine by reference. When the C routine finishes, the contents of this object are returned as one of the values to the calling function. If the object returned is a record or array, it will be copied to a new alien value which will be returned.
:copy	This is similar to :in, but the argument is copied to a pre-allocated object and a pointer to this object is passed to the C routine.
:in-out	A combination of :copy and :out. The argument is copied to a pre-allocated object and a pointer to this object is passed to the C routine. On return, a new alien value is allocated for the object and returned as a multiple value.

For example, the C function cfoo with the following calling conventions:

```
cfoo (a, i)
     char a;
     int i;
{
/* Body of cfoo. */
}
```

can be described by the following call to def-c-routine:

```
(def-c-routine ("cfoo" lfoo) (void)
        (a char)
        (i int))
```

10.6. Calling Lisp routines from C

It is sometimes necessary to pass a procedure pointer to a C routine so that at some later time C code can call the procedure. An example of this is menus, where associated with each menu item is a procedure to call when that menu item is selected. A simple mechanism has been built into the foreign function interface to make it possible to pass an object which looks like a C procedure pointer into C from Lisp. When this procedure object is invoked a Lisp function will be called instead of normal C code.

extensions:def-c-procedure *symbol nargs function* [*Macro*]

Def-c-procedure sets the value of *symbol* to a structure that can be passed as a parameter to a foreign function that expects a pointer to a procedure object. *Nargs* should be the number of parameters the procedure is going to be called with. *Function* should be a lisp object which can be invoked by apply. *Function* should accept the number of arguments specified by *nargs*, if not an error is signalled. When C calls this procedure object, *function* will gain control. There are no restrictions on what this function can do. It may call other C routines, throw to a catch tag above where C code was initially invoked, invoke the Lisp error system, etc.

For example, if you have the following C code:

```
calllisp(p)
    int (*p)();
{   int i;

    i = (*p)(5);
    printf("I = %d.\n", i);
}
```

You can invoke it from Lisp as follows:

```
(def-c-routine "calllisp" (void) (p system:c-procedure))
(def-c-procedure foo 1 #'(lambda (x) (+ x 5)))
(calllisp foo)
```

If you do (calllisp foo) outside of Hemlock, since the C code is doing output, you should get the following results:

```
I = 10.
```

10.7. An Example

This section presents a complete example of an interface to a somewhat complicated C function. This example should give a fairly good idea of how to get the effect you want for almost any kind of C function.

Suppose you have the following C function which you want to be able to call from Lisp in the file cfun.c:

```
struct cfunr {
    int x;
    char *s;
};

struct cfunr *cfun (i, s, r, a)
    int i;
    char *s;
    struct cfunr *r;
    int a[10];
{   int j;
    struct cfunr *r2;

    printf("i = %d\n", i);
    printf("s = %s\n", s);
    printf("r->x = %d\n", r->x);
    printf("r->s = %s\n", r->s);
    for (j = 0; j < 10; j++) printf("a[%d] = %d.\n", j, a[j]);
    r2 = (struct cfunr *) malloc (sizeof(struct cfunr));
    r2->x = i + 5;
    r2->s = "A C string";
    return(r2);
};
```

It is possible to call this function from Lisp using the file cfun.lisp whose contents is:

```
;;; -*- Package: test-c-call; Mode: Lisp -*-
(in-package "TEST-C-CALL" :use '("LISP" "SYSTEM" "EXTENSIONS"))

;;; Define c-string as a null-terminated string of up to 256 characters.
(def-c-type c-string (null-terminated-string 256))

;;; Define a *c-string to be a pointer to a c-string.
(def-c-pointer *c-string c-string)

;;; Define the record cfunr in Lisp.
(def-c-record cfunr
        (x int)
        (s *c-string))

;;; Define the C array ar to have 10 elements of type int.
(def-c-array ar int 10)

;;; Define the C type pointer to the array above.
(def-c-pointer *ar ar)
;;; Load in the C object file with the function definition.
(load-foreign "cfun.o")

;;; Define the Lisp function interface to the C routine.  It returns a
;;; pointer to a record of type cfunr.  It accepts four parameters: i,
;;; an int; s, a pointer to a string; r, a pointer to a cfunr record;
;;; and a, a pointer to the array defined above.
(def-c-routine "cfun" (*cfunr)
        (i int)
        (s *c-string)
        (r *cfunr)
        (a *ar))

;;; A function which sets up the parameters to the C function and
;;; actually calls it.
(defun call-cfun ()
  (let ((arr (make-ar))                     ; Make an array.
        (rec (make-cfunr)))                 ; Make a record.
    (alien-bind ((a arr ar t)
                 (r rec cfunr t))
      (dotimes (i 10)                       ; Fill array.
        (setf (alien-access (ar-ref (alien-value a) i)) i))
      (setf (alien-access (cfunr-x (alien-value r))) 20)
      (setf (alien-access (cfunr-s (alien-value r)) 'pointer)
            "A Lisp String")
      (let ((rec2 (cfun 5 "Another Lisp String"
                        (alien-sap (alien-value r))
                        (alien-sap (alien-value a)))))
        (format t "Returned from C function.~%")
        (alien-bind ((r2 rec2 cfunr t))
          (let ((cs (alien-access (cfunr-s (alien-value r2)) 'alien)))
            (alien-bind ((s cs (null-terminated-string 256) t))
              (values (alien-access (cfunr-x (alien-value r2)))
                      (alien-access (alien-value s))))))))))
```

To execute the above example, it is necessary to compile the c routine as follows:

```
cc -c cfun.c
```

Once this has been done, you should start up lisp, and do the following:

```
lisp
;;; Lisp should start up with its normal prompt.

;;; Next compile the lisp file
* (compile-file "cfun.lisp")
Error output from cfun.lisp 17-Mar-87 17:09:57.
Compiled on 18-Mar-87 17:33:16 by CLC version M1.6 (16-Mar-87).

INDIRECT-*C-STRING compiled.
MAKE-CFUNR compiled.
INDIRECT-*CFUNR compiled.
CFUNR-X compiled.
CFUNR-S compiled.
MAKE-AR compiled.
AR-REF compiled.
INDIRECT-*AR compiled.
CFUN compiled.
Warning in CALL-CFUN:
  Could not show 32 bit store to be word-aligned:
(AR-REF A I)
CALL-CFUN compiled.

Finished compilation of file "/usr/dbm/cfun.lisp".
0 Errors, 1 Warnings.
Elapsed time 0:00:10, run time 0:00:09.

;;; Now load the file:
* (load "cfun")
;;; Lisp prints out the following information:
[Loading foreign files (cfun.o) ...
  [Running ld ... done.]
  [Reading Unix object file ... done.]
  [Loading symbol table information ... done.]
done.]
T
```

```
;;; Now call the routine that sets up the parameters and calls the C
;;; function.
* (test-c-call::call-cfun)
;;; The C routine prints the following information to standard output.
i = 5
s = Another Lisp string
r->x = 20
r->s = A Lisp string
a[0] = 0.
a[1] = 1.
a[2] = 2.
a[3] = 3.
a[4] = 4.
a[5] = 5.
a[6] = 6.
a[7] = 7.
a[8] = 8.
a[9] = 9.
;;; Lisp prints out the following information.
Returned from C function.
;;; Return values from the call to test-c-call::call-cfun.
10
"A C string"
*
```

If any of the foreign functions do output, they should not be called from within Hemlock. Depending on the situation, various strange behavior occurs. On the console, you will see no output; under X, the output goes to the window in which Lisp was started; on a terminal, the output will be placed in the current buffer but will not be recognized by Hemlock. This means it will overwrite information already in the window and be overwritten by Hemlock. This will not have any impact on the contents of the buffer, since the output is coming from a source that Hemlock does not know about.

Chapter 11

User-Defined Assembler Language Routines

By David B. McDonald

11.1. Introduction

The CMU Common Lisp implementation on the IBM RT PC has been modified to make it relatively easy for a user to write assembler language routines (miscops) that can be executed from Lisp. It is important to note that a miscop has access to the state of the Lisp system, and thus has the potential of clobbering the Lisp beyond recovery. In particular, there are several conventions used in coding miscops that must be adhered to. These conventions and restrictions are described in a later section of this chapter.

11.2. Notation

The IBM RT PC numbers the bits of a register differently from many other machines. Bit 0 is the leftmost bit, and bit 31 is the right most bit. The notation C0 specifies the leftmost byte of a register. The notation C3 specifies the rightmost byte of a register.

11.3. Defining User Miscops

All the assembler instructions are internal to the compiler package. To define miscops, you should have a file that has only miscops in it. The first form before any miscops should be:

```
    (in-package 'compiler)
```

This will give you access to the assembler instructions, as well as some macros and constants that will aid you in writing miscops.

Defining a miscop is easy, just do the following:

```
    (define-user-miscop name
            A1
            A2
            . . .
            AN
    )
```

Where name is the name of the miscop you are defining. The A_i are one of the following:

- A keyword which becomes an external label that you can branch to from another miscop as well as the current one.
- A symbol which becomes a label which you can branch to from somewhere else in the current miscop.

- A list which is either a IBM RT PC assembler instruction or a call to a macro. If it is a macro, the macro is expanded, and the assembler splices the resulting list into the instruction stream and starts assembling it. If it is an IBM RT PC instruction, then it is just assembled. Note that macros can expand into calls to other macros.

11.4. The Assembler

`compiler:assemble-file` *input-pathname* **`&key :output-file :error-file :listing-file :unixy-lap-file`** [*Function*]

Assemble-file assembles the miscops contained in the file *input-pathname* with default extension **romp**. The **`:output-file`** argument specifies where the assembled code should go. The default is the same name as *input-pathname* with extension **fasl**. The **`:error-file`** argument specifies where the error messages should be printed. The default is the same name as *input-pathname* with extension **err**. The argument **`:listing-file`** specifies where a listing of the code generated should go. The default is not to generate a listing file. If specified as T, the name of the listing file is the same name as *input-pathname* with extension **list**. The argument **`:unixy-lap-file`** specifies that the listing file (with extension s) should be a file acceptable to a Unix assembler.

The assembler accepts instructions in the following format:

`(opcode` O_1 O_2 **`...`** O_N**`)`**

Where opcode is a mnemonic for one of the IBM RT PC instructions, and the O_i are the operands to the instruction. These operands take one of several forms:

- It may be a register, which should be specified as one of the following:

Register	Number	Normal Use
NL0	0	Non-Lisp Temporary.
A0	1	First argument to miscop.
NL1	2	Non-Lisp Temporary.
A1	3	Second argument to miscop.
A3	4	Fourth argument to miscop.
A2	5	Third argument to miscop.
CS	6	Control stack pointer.
L0	7	Lisp function local 0.
L1	8	Lisp function local 1.
L2	9	Lisp function local 2.
L3	10	Lisp function local 3.
L4	11	Lisp function local 4.
BS	12	Binding stack pointer.
FP	13	Frame pointer for current function.
AF	14	Active function pointer for current function.
PC	15	Return PC.

  Registers 5 and below can be destroyed by the miscop. The return value for the miscop should be left in A0. Only one value may be returned by a miscop. There is no way to return multiple values directly from a miscop. Registers 6 through 14 must either be untouched or restored to the correct value when the miscop returns. Register 15 contains the return address back to compiled Lisp code. If any arguments are passed on the stack, they must be popped off the stack before returning to compiled Lisp code.

- It may be a fixnum in which case that value is used as the operand.
- It may be a Lisp expression, in which case it is evaluated and the resulting fixnum value is used as the operand.
- If the opcode is a branch instruction, then the branch target should be a symbol which is a label. If a label is not defined in the current miscop, it is assumed to be external by the assembler. This allows

you to call other miscops from within a miscop.

11.5. Assembler Instructions

This section contains a list of all the IBM RT PC instructions supported by the assembler. The meaning of the symbols used in the instruction definitions are:

- R - a register, may be indexed if more than one register is required by the instruction.
- 0/(R) - a register, however, if the register is register 0, use the value 0, otherwise use the contents of the register. This is useful in certain circumstances where you don't want to use a base register.
- I4 - an immediate value, may be a fixnum or a Lisp expression which will be evaluated and must evaluate to a fixnum. Only the low order 4 bits of the fixnum are significant.
- I16 - as I4, but the low order 16 bits are significant. This is an unsigned value.
- SI16 - as I16, but signed.
- L - a label, which should be a symbol that must be a label in the current miscop or exist as an external label when labels are resolved at load time.

The IBM RT PC supports several classes of instructions. These will be dealt with in separate sub-sections below.

11.5.1. Storage Access

The storage access instructions are used to refer to memory. These instructions can load or store the contents of registers from or to memory. To perform any operation, it is necessary to get the data into a register. None of these instructions set the condition codes.

Load character short: **(lcs R_1 R_2 I4)**
: The byte addressed by R_2 + I4 is loaded into the low order byte R_1 and the high order bytes are zeroed.

Load character: **(lc R_1 R_2 SI16)**
: Similar to **lcs**, except the offset is a signed 16 bit quantity.

Load half algebraic short: **(lhas R_1 R_2 I4)**
: The signed 16 bit quantity addressed by R_1 + 2*I4 is signed extended and loaded into register R_1.

Load half algebraic: **(lha R_1 R_2 SI16)**
: Similar to **lha**, except the offset is not shifted and is a signed 16 bit quantity.

Load half short: **(lhs R_1 R_2)**
: The unsigned 16 bit quantity addressed by R_2 is loaded into R_1. The upper half word of R_1 is set to zero.

Load half: **(lh R_1 R_2 SI16)**
: The unsigned 16 bit quantity addressed by R_2 + SI16 is loaded into the low half word of R_1. The upper half of R_1 is cleared.

Load short: **(ls R_1 R_2 I4)**
: The word addressed by R_2 + 4 * I4 is loaded into R_1.

Load: **(l R_1 R_2 SI16)**
: Load is similar to load short, except the offset is a signed 16 bit quantity.

Load multiple: **(lm R_1 R_2 SI16)**
: Load multiple loads the word at R_2 + SI16 into register R_1, loads the word at R_2 + SI16 + 4 into R_1 + 1, etc. This process continues until R15 has been loaded.

Test and set half: **(tsh R_1 R_2 SI16)**

The upper half of R_1 is set to 0, the lower half is set to the half word addressed by R_2 + SI16. Immediately after reading the half word, the upper byte of the half word is replaced by 1's. The lower byte is left unchanged.

Store character short: **(stcs R_1 R_2 I4)**
The character addressed by R_2 + I4 is replaced by the low order byte of R_1.

Store character: **(stc R_1 R_2 SI16)**
is similar to store character short, except the offset is a signed 16 bit quantity.

Store half short: **(sths R_1 R_2 I4)**
The half word addressed by R_2 + 2 * I4 is replaced by the low order half word of R_1.

Store half: **(sth R_1 R_2 SI16)**
The half word addressed by R_2 + SI16 is replaced by the low order half word of R_1.

Store short: **(sts R_1 R_2 I4)**
The word addressed by R_2 + 4 * I4 is replaced by the contents of R_1.

Store: **(st R_1 R_2 SI16)**
The word addressed by R_2 + SI16 is replaced by the contents of R_1.

Store multiple: **(stm R_1 R_2 SI16)**
Store multiple stores the contents of R_1 into the word addressed by R_2 + SI16, stores the contents of R_1 + 1 into the word addressed by R_2 + SI16 + 4, etc. This process continues until R15 has been stored.

Note that the IBM RT PC is a byte addressed machine. When addressing a half word, the low order bit of the address should be 0. If it is not, then the low order bit is forced to 0, and then the operation is performed. This can cause some strange results. Note also, that the short instructions use the 4 bit immediate field differently depending on the size of the storage element being referenced. It refers to the I'th element of that particular size. The long instructions do not follow this convention, they just use the immediate value as it is.

11.5.2. Address Computation

The address computation instructions operate only on registers. None of these instructions set the condition codes.

Compute address lower half: **(cal R_1 R_2 SI16)**
The value 0/(R_2) + SI16 is placed in register R_1.

Compute address lower half 16-bit: **(cal16 R_1 R_2 I16)**
The low order half of R_1 is replaced by 0/(R_2) + I16. The upper half of R_1 is replaced by the upper half of 0/(R_2).

Compute address upper half: **(cau R_1 R_2 I16)**
The low order half of R_1 is replaced by the low half of 0/(R_2). The high order half of R_1 is replaced by the high half of 0/(R_2) + I16.

Compute address short: **(cas R_1 R_2 R_3)**
Register R_1 is replaced by the contents of R_2 plus 0/(R_3).

Compute address 16 bit: **(ca16 R_1 R_2)**
The low order halves of R_1 and R_2 are added together and replace the low order half of R_1. The high order half or R_1 is replaced by the high half of R_2.

Increment: **(inc R_1 I4)**
The immediate value I4 is added to the contents of R_1.

Decrement: **(dec R_1 I4)**
The immediate value I4 is subtracted from the contents of R_1.

Load immediate short: **(lis R_1 I4)**
The immediate value I4 replaces the contents of R_1.

11.5.3. Branching

For most of the branching instructions, there are two forms. One form is the standard form and just branches normally. The other form is an execute form which executes the following instruction at the same time a branch is taken. If a branch is not taken, the following instruction is executed in the normal sequence. For the branch and link with execute instructions, the address stored in the link register is 4 bytes beyond the current instruction. This means that if the following instruction is only a 2 byte instruction, a 2 byte noop instruction must be inserted, so that the correct instruction is returned to. Some instructions can not be the target of a branch with execute instruction. These instructions include all the branch instructions, all the trap instructions, the load program status instruction, the supervisor call instruction, and the wait instruction. Those instructions which have an execute counterpart are specified by a trailing [execute] in the name and a trailing [x] in the instruction mnemonic.

None of these instructions alter the condition code bits.

Branch and link absolute [execute]: **(bala[x] I24)**
: The 24 bit immediate field is used as an address and control is transferred to that address. At the same time, the address of the next instruction [+ 4] is stored in register R15.

Branch and link immediate [execute]: **(bali[x] R_1 L)**
: The label L is represented as an offset from the bali[x] instruction and thus has to be within range. However, this is a large range, and if you manage to get outside of this range, you have written too much assembler code by at least two orders of magnitude. The address of the next instruction [+ 4] is placed in R_1. This is the instruction you should use to call other miscops.

Branch and link [execute]: **(balr[x] R_1 R_2)**
: R_2 should contain the address of some code. Control is passed to this address. The address of the next instruction [+ 4] is placed in R_1.

Branch condition immediate: **(b*cc*[x] L)**
: The *cc* specifies the condition to branch on. The legal values are:

<none>	Unconditional branch.
eq	Branch if eq condition bit is set.
ne	Branch if eq condition bit is not set.
lt	Branch if lt condition bit is set.
gt	Branch if gt condition bit is set.
ge	Branch if lt condition bit is not set.
le	Branch if gt condition bit is not set.
[n]ov	Branch if the overflow bit is [not] set.
[n]tb	Branch if the test bit is [not] set.
[n]c0	Branch if the carry bit is [not] set.

If the execute form is used and the branch is taken, the next instruction is executed while the instruction at the target address is being fetched from memory.

Branch condition: **(br*cc*[x] R_1)**
: This instruction is similar to the branch condition immediate instruction, except the target address is in a register. The *cc* have the same meaning as above. The instruction:

```
    brx      PC
    <last instruction of miscop>
```

should be used to return to lisp code from a miscop.

11.5.4. Traps

The trap instructions cause an exception to be generated if the condition associated with the trap instruction is not met. None of these instructions affect the condition code bits.

Trap on condition immediate: **(ti I4 R_1 SI16)**
: The value of R_1 is compared with the sign extended 16 bit value SI16. The I4 field specifies the condition on which the trap is enabled:

8	Trap if the value in R_1 is less than SI16.
4	Trap if the value in R_1 is equal to SI16.
2	Trap if the value in R_1 is greater than SI16.

These values can be ored together to get more than one trap condition.

Trap if register greater than or equal: **(tgte R_1 R_2)**
If the contents of R_1 is greater than or equal to R_2, a trap occurs.

Trap if register less than: **(tlt R_1 R_2)**
If the contents of register R_1 is less than R_2, a trap occurs.

11.5.5. Moves and inserts

These instructions move data between registers, and between registers and the test bit of the condition code. Except for the test bit, none of the condition code bits are altered.

Move character zero from three: **(mc03 R_1 R_2)**
Byte C0 of R_1 is replaced by byte C3 of R_2.

Move character one from three: **(mc13 R_1 R_2)**
Byte C1 of R_1 is replaced by byte C3 of R_2.

Move character two from three: **(mc23 R_1 R_2)**
Byte C2 of R_1 is replaced by byte C3 of R_2.

Move character three from three: **(mc33 R_1 R_2)**
Byte C3 of R_1 is replaced by byte C3 of R_2.

Move character three from zero: **(mc30 R_1 R_2)**
Byte C3 or R_1 is replaced by byte C0 of R_2.

Move character three from one: **(mc31 R_1 R_2)**
Byte C3 of R_1 is replaced by byte C1 of R_2.

Move character three from two: **(mc32 R_1 R_2)**
Byte C3 of R_1 is replaced by byte C2 of R_2.

Move from test bit: **(mftb R_1 R_2)**
The bit of R_1 specified by bits 27-31 of R_2 is set to the value of the test bit in the condition code.

Move from test bit immediate lower half: **(mftbil R_1 I4)**
The bit of the lower half of R_1 specified by I4 is set to the value of the test bit in the condition code.

Move from test bit immediate upper half: **(mftbiu R_1 I4)**
The bit of the upper half of R_1 specified by I4 is set to the value of the test bit in the condition code.

Move to test bit: **(mttb R_1 R_2)**
The test bit of the condition code is set to the bit of R_1 specified by the value of bits 27-31 of R_2.

Move to test bit immediate lower half: **(mttbil R_1 I4)**
The test bit of the condition code is set to the value of bit in the lower half of R_1 specified by I4.

Move to test bit immediate upper half: **(mttbiu R_1 I4)**
The test bit of the condition code is set to the value of bit in the upper half of R_1 specified by I4.

11.5.6. Arithmetic Operations

The arithmetic instructions set various condition code bits. The description of each instruction gives the set of condition code bits that are set by the instruction.

Add: **(a R_1 R_2)** The contents of R_2 is added to the contents of R_1 leaving the result in R_1. Condition code bits lt, eq, gt, c0, and ov are modified.

Add extended: **(ae R_1 R_2)**
The contents of R_2 plus the value of the carry bit is added to the contents of R_1 leaving the result in R_1. Condition code bits lt, eq, gt, c0, and ov are modified.

Add extend immediate: **(aei R_1 R_2 SI16)**
The contents of R_2 plus SI16 plus the value of the carry bit replaces the contents of R_1. Condition code bits lt, eq, gt, c0, and ov are modified.

Add immediate: **(ai R_1 R_2 SI16)**
The contents of R_2 plus SI16 replaces the contents of R_1. Condition code bits lt, eq gt, c0, and ov are modified.

Add immediate short: **(ais R_1 I4)**
The value I4 is added to R_1. Condition code bits lt, eq, gt, c0, and ov are modified.

Absolute: **(abs R_1 R_2)**
The absolute value of R_2 replaces the contents of R_1. Condition code bits lt, eq, gt, c0, and ov are modified.

Ones complement: **(onec R_1 R_2)**
The ones complement of R_2 replaces the contents of R_1. Condition code bits lt, eq, and gt are modified.

Twos complement: **(twoc R_1 R_2)**
The twos complement of R_2 replaces the contents of R_1. Condition code bits lt, eq, gt, c0, and ov are modified.

Compare: **(c R_1 R_2)**
The signed twos complement numbers in R_1 and R_2 are compared. The lt bit is set to 1 if R_1 is less than R_2, the eq bit is set if R_1 is equal to R_2, and the gt bit is set if R_1 is greater than R_1.

Compare immediate short: **(cis R_1 I4)**
The signed twos complement number in R_1 is compared to the immediate value I4. Condition code bits lt, eq, and gt are modified as above.

Compare immediate: **(ci R_1 SI16)**
The signed twos complement number in R_1 is compared to the signed number SI16. Condition code bits lt, eq, and gt are modified as above.

Compare logical: **(cl R_1 R_2)**
The unsigned numbers in R_1 and R_2 are compared for magnitude only. Condition code bits lt, eq, and gt are modified.

Compare logical immediate: **(cli R_1 I16)**
The unsigned 32 bit number in R_1 is compared to the unsigned 16 bit immediate value I16 extended to the left with 16 0's. Condition code bits lt, eq, and gt are modified.

Extend sign: **(exts R_1 R_2)**
The lower half of R_2 replaces the lower half of R_1. Bit 16 (the sign bit of the half word) of R_2 replaces bits 0-15 of R_1. Condition code bits lt, eq, and gt are modified.

Subtract: **(s R_1 R_2)** The contents of R_2 is subtracted from the contents of R_1 leaving the result in R_1. Condition code bits lt, eq, gt, c0, and ov are modified.

Subtract from: **(sf R_1 R_2)**
The contents of R_1 is subtracted from the contents of R_2 leaving the result in R_1. Condition code bits lt, eq, gt, c0, and ov are modified.

Subtract extended: **(se R_1 R_2)**
The ones complement of R_2 is added to the contents of R_1. This result is added to the value of the c0 condition code bit. The result is placed in R_1. Condition code bits lt, eq, gt, c0, and ov are modified.

Subtract from immediate: **(sfi R_1 R_2 SI16)**
The contests of R_2 is subtracted from SI16 leaving the result in R_1. Condition code bits lt, eq,

gt, c0, and ov are modified.

Subtract immediate short: **(sis R_1 I4)**
The value I4 is subtracted from R_1. Condition code bits lt, eq, gt, c0, and ov are modified.

Divide step: **(d R_1 R_2)**
If you really want to do a divide step see *IBM RT PC Hardware Technical Reference Manual.* See section 11.7 for a miscop routine that does division.

Multiply step: **(m R_1 R_2)**
Again, you don't really want to use this instruction. See 11.7 for a miscop that does multiplication.

11.5.7. Logical Operations

The logical operations treat the registers as 32 bit unsigned quantities. Condition code bits are set according to the result as a 32 bit twos complement number.

Clear bit lower half: **(clrbl R_1 I4)**
The bit specified by I4 in the lower half of R_1 is set to 0.

Clear bit upper half: **(clrbu R_1 I4)**
The bit specified by I4 in the upper half of R_1 is set to 0.

Set bit lower half: **(setbl R_1 I4)**
The bit specified by I4 in the lower half of R_1 is set to 1.

Set bit upper half: **(setbu R_1 I4)**
The bit specified by I4 in the upper half of R_1 is set to 1.

And: **(n R_1 R_2)** The logical and of R_1 and R_2 replaces the contents of R_1.

And immediate lower half extended zeroes: **(nilz R_1 R_2 I16)**
The logical and of R_2 and I16 extended on the left by 16 0's replaces the contents of R_1.

And immediate lower half extended ones: **(nilo R_1 R_2 I16)**
The logical and of R_2 and I16 extended on the left by 16 1's replaces the contents of R_1.

And immediate upper half extended zeroes: **(niuz R_1 R_2 I16)**
The logical and of R_2 and I16 extended on the right by 16 0's replaces the contents of R_1.

And immediate upper half extended ones: **(niuo R_1 R_2 I16)**
The logical and of R_2 and I16 extended on the right by 16 1's replaces the contents of R_1.

Or: **(o R_1 R_2)** The logical or of R_1 and R_2 replaces the contents of R_1.

Or immediate lower: **(oil R_1 R_2 I16)**
The logical or of R_2 and I16 extended on the left by 16 0's replaces the contents of R_1.

Or immediate upper: **(oiu R_1 R_2 I16)**
The logical or of R_2 and I16 extended on the right by 16 0's replaces the contents of R_1.

Exclusive or: **(x R_1 R_2)**
The logical exclusive or of R_1 and R_2 replaces the contents of R_1.

Exclusive or immediate lower half: **(xil R_1 R_2 I16)**
The logical exclusive or of R_2 and I16 extended on the left by 16 0's replaces the contents of R_1.

Exclusive or immediate upper half: **(xiu R_1 R_2 I16)**
The logical exclusive or of R_2 and I16 extended on the right by 16 0's replaces the contents of R_1.

Count leading zeroes: **(clz R_1 R_2)**
The contents of R_1 is replaced by count of the leading zeroes in the low half word of R_2. If the low half of R_2 is 0, R_1 is set to 16.

11.5.8. Shifts

Shift instructions set the condition code bits lt, eq, and gt according to the result of the shift as a twos complement number. For shift amounts in registers, the low order 6 bits are used as a shift count. Except for the algebraic shifts, 0's are shifted into vacated bits of a register. For algebraic shifts, the sign bit replaces vacated bits.

Shift algebraic right: **(sar R_1 R_2)**
The contents of register R_1 is shifted right by the amount specified in R_2. The original sign of R_1 replaces any bits vacated by the shift.

Shift algebraic right immediate: **(sari R_1 I4)**
The contents of register R_1 are shifted right by the amount specified by I4.

Shift algebraic right immediate plus sixteen: **(sari16 R_1 I4)**
The contents of register R_1 are shifted right by the amount specified by I4 + 16.

Shift right: **(sr R_1 R_2)**
The contents of register R_1 is shifted right by the amount specified by R_2.

Shift right immediate: **(sri R_1 I4)**
The contents of register R_1 is shifted right by the amount specified by I4.

Shift right immediate plus sixteen: **(sri16 R_1 I4)**
The contents of register R_1 is shifted right by the amount specified by I4 + 16.

Shift right paired: **(srp R_1 R_2)**
The value in R_1 is shifted right by the amount specified by R_2. The result is placed in the twin of R_1 rather than R_1. Each even/odd set of registers are paired. The twin of an even register is the odd one of the pair, similarly for the odd register.

Shift right paired immediate: **(srpi R_1 I4)**
is similar to srp, except the shift amount is specified by I4.

Shift right paired immediate plus sixteen: **(srpi16 R_1 I4)**
is similar to srp, except the shift amount is specified by I4 + 16.

Shift left: **(sl R_1 R_2)**
The contents of R_1 is shifted left by the amount specified by R_2.

Shift left immediate: **(sli R_1 I4)**
The contents of R_1 is shifted left by the amount specified by I4.

Shift left immediate plus sixteen: **(sli16 R_1 I4)**
The contents of R_1 is shifted left by the amount specified by I4 + 16.

Shift left paired: **(slp R_1 R_2)**
is similar to shift right paired, except the shift is to the left.

Shift left paired immediate: **(slpi R_1 I4)**
is similar to slp, except the shift amount is specified by I4.

Shift left paired immediate plus sixteen: **(slpi16 R_1 I4)**
is similar to slpi, except the shift amount is specified by I4 + 16.

11.5.9. System Control

Move to SCR: **(mts R_1 R_2)**
The contents of system control register R_1 is replaced by R_2.

Move from SCR: **(mfs R_1 R_2)**
The contents of system control register R_1 replaces the contents of R_2.

Clear SCR bit: **(clrsb R_1 I4)**
The bit specified by I4 of the lower half of system control register R_1 is set to 0.

Set SCR bit: **(setsb R_1 I4)**

The bit specified by I4 of the lower half of system control register R_1 is set to 1.

Load program status: **(lps R_1 R_2 SI16)**
This is here for completeness. You should never use this instruction from within Lisp. This is a privileged instruction and should cause an exception.

Wait: **(wait)** Puts the processor in wait state. However, this is privileged instruction, and should cause an exception.

Supervisor call: **(svc 0 R_1 SI16)**
The lower order 16 bits of 0/(R_1) + SI16 specifies a system call code. The host operating system gains control and performs some operation. There should be no need for you to use this instruction.

11.5.10. Input/Output

These instructions are privileged and should not be used from Lisp.

Input/output read: **(ior R_1 R_2 I16)**
The contents of R_1 is replaced by data transferred from an IO device specified by 0/(R_2) + I16.

Input/output write: **(iow R_1 R_2 I16)**
The contents of R_1 are transferred to the IO device specified by 0/(R_2) + I16.

11.6. Useful Macros

This section contains a set of useful macros that have been developed to make writing miscops easier.

11.6.1. Saving and Restoring Registers

(save-registers R_1 ... R_n)
The register R_1, ..., R_n are saved on the stack. If you need to use more registers than the first six registers, you must save them first.

(restore-registers R_1 ... R_n)
The registers R_1, ..., R_n are restored from the stack. There should be a matching save-registers call. The arguments to both macros should be **exactly** the same.

(save-registers-pc R_1 ... R_n)
is similar to save-registers, except that the PC register to return to Lisp code is also saved on the stack. Note that if no registers are specified, just the PC is saved on the stack.

(restore-registers-pc R_1 ... R_n)
is similar to restore-registers, except that the PC register to return to Lisp code is restored correctly.

(save-registers-internal-pc R_1 ... R_n)
is similar to save-registers-pc, except the PC register contains a return address in miscop space rather than in Lisp code space.

(restore-registers-internal-pc R_1 ... R_n)
is similar to restore-registers-pc, except it restores an address to a miscop.

11.6.2. Storage Allocation

(allocate register type length temp1 temp2)
allocates a Lisp object from the current allocation space. A pointer to the resulting object is placed in **register**. **Type** specifies the type of the object and must be one of the symbols: type-bignum, type-ratio, type-long-float, type-complex, type-string, type-bit-vector, type-i-vector, type-g-vector, type-array, type-function, type-symbol, or type-list. **Length** specifies the

length of the object to allocate and may be either a constant or a register. The length must include space for any header. The following symbols specify the lengths of some of the more common objects: bignum-header-size, long-float-size, string-header-size, bit-vector-header-size, i-vector-header-size, g-vector-header-size, array-header-size, function-header-size, symbol-size, cons-size. **Temp1** and **temp2** are two temporary registers. These registers can not be NL0. Note that this only allocates the storage, it does not set up any headers or store information into the object. You must do this yourself. See the document *Internal Design of Common Lisp on the IBM RT PC* for the format of Lisp objects. Also note that this macro does not check to see if a garbage collection should be done. Most of the miscops that perform allocation must do this themselves. If you are allocating large amounts of storage without doing any computations from Lisp you may run out of storage. If you think you might be having this trouble, the following sequence of code should be used to exit a miscop that is returning a newly allocated object in register A0.

```
(lr      NL0 A0)
<subtract length allocated from NL0>
(x       NL0 A0)
(niuz    NL0 NL0 #xFFFE)
(breq    PC)
(b       maybe-gc)
```

The above code will check to see if the allocation went over a 64K boundary. If it did, then it may be time to GC. Maybe-gc is a miscop that calls out to Lisp to check if it is time to GC. The Lisp Maybe-gc function is passed the object in A0 and returns it as if the miscop returned.

(static-allocate register type length temp1 temp2)
Static-allocate is similar to allocate, except that it allocates storage in static space rather than the space specified by current-allocation-space.

11.6.3. Error reporting

There are three macros that you can use to invoke the Lisp function %SP-Internal-Error. %SP-Internal-Error will report the error to the user.

(error0 error-code)
This macro invokes %SP-Internal-Error with no optional arguments. **Error-code** should be a literal fixnum specifying the error code. See the manual *Internal Design of Common Lisp on the IBM RT PC* for a list of the current error codes.

(error1 error-code object)
is similar to error0, except one optional argument is passed to %SP-Internal-Error. This argument should be the object that has caused the error. For example, if you are expecting a symbol, and get something else, then the something else would be passed out to %SP-Internal-Error. **Object** should be a register containing the object in question.

(error2 error-code object$_1$ object$_2$)
is similar to error1, except two arguments are passed to %SP-Internal-Error.

11.6.4. Type Checking

Several macros are provide that check the types of objects in registers. These macros make assumptions about the register usage. In particular, register NL0 and/or NL1 may be destroyed by these macros. These macros are normally used on miscop entry and thus NL0 and NL1 will contain nothing important. You must follow this convention if you want to use these macros.

(verify-type register type error &optional ignore-nil)
This macro verifies that **register** contains an object of type **type**. If it does not, then the macro generates code that will branch to the label **error**. The label **Error** need not be defined in the current file, but it must be defined when references are resolved. The optional argument **ignore-nil** is used when **type** is type-symbol. If it is non-nil, NIL is not valid as a symbol.

(verify-not-type register type error)

This macro is similar to verify-type, except that **register** should not contain an object of **type**.

(test-nil register label)
branches to **label** if **register** contains NIL.

(test-not-nil register label)
branches to **label** if **register** does not contain NIL.

(test-t register label)
branches to **label** if **register** contains T.

(test-not-t register label)
branches to **label** if **register** does not contain T.

(test-trap register label)
branches to **label** if **register** contains the trap object.

(test-not-trap register label)
branches to **label** if **register** does not contain the trap object.

(get-type register type-register)
extracts the type code from the object in **register** and places the five bit type code in **type-register** zeroing the high order bits. Note that this macro generates best code when **register** is A0 and **type-register** is NL0 or **register** is A1 and **type-register** is NL1.

(type-equal register type label)
branches to **label** if **register** contains the type code for **type**.

(type-not-equal register type label)
is similar to type-equal, except it branches to **label** if **register** does not contain the type code for **type**.

11.6.5. Miscellaneous

(noop) This macro generates a two byte instruction which does absolutely nothing. This is often used after a branch and link with execute instruction if the executed instruction is a two byte one.

(pushm register) pushes the contents of **register** onto the control stack.

(popm register) pops the top of the control stack into **register**.

(lr R_1 R_2) The contents of R_2 is copied to R_1.

(loadi R_1 I) The immediate value I is loaded into R_1 using the best sequence of code. Up to a 32 bit number can be loaded with this macro.

(cmpi R_1 I) The value in R_1 is compared with the immediate value I using the appropriate instruction. The I value can be a 16 bit signed number.

(loadc R_1 R_2 &optional (offset 0))
Uses the short or long form of the load character instruction depending on the value of offset.

(loadha R_1 R_2 &optional (offset 0))
Uses the short or long form of the load halfword algebraic instruction depending on the value of offset.

(loadh R_1 R_2 &optional (offset 0))
Uses the short or long form of the load halfword instruction depending on the value of offset.

(loadw R_1 R_2 &optional (offset 0))
Uses the short or long form of the load instruction depending on the value of offset.

(storec R_1 R_2 &optional (offset 0))
Uses the short or long form of the store character instruction depending on the value of offset.

(storeha R_1 R_2 &optional (offset 0))
Uses the short or long form of the store halfword instruction depending on the value of offset.

(storew R_1 R_2 &optional (offset 0))

Uses the short or long form of the store instruction depending on the value of offset.

(multiply R_1 R_2) multiplies R_1 by R_2 leaving the high order result in R_1 and the low order result in R_2. Note that this is a 32 bit by 32 bit multiply.

11.6.6. Floating Point

The following macros allow you to access the floating point accelerator card from a miscop. You should see the manual *IBM RT PC Hardware Technical Reference Manual* for more information about the floating point card.

There are two floating point formats: a short (or single) format using 32 bits and a double (or long) format using 64 bits. There are sixteen 32-bit floating point registers accessible to Lisp. These registers are never saved by Lisp, since they are only modified during the execution of miscops. The Lisp miscops return the resulting value in Lisp format and never leaves information in the floating point registers. Register 14 and 15 of the floating point set are reserved for special purposes. The other fourteen can be used by a miscop for whatever purpose it needs. For 64 bit floats, the even register of a pair contains the high order data (including exponent) and the odd one of the pair contains the low order data (the least significant bits of the mantissa).

In the following descriptions, **gr** stands for a general purpose register, **fr** stands for a floating point register, and **base** is a general register used to provide addressability to the floating point accelerator card. If more than one general or floating point register is needed in an instruction, the **gr**'s and **fr**'s are numbered.

(rdfr gr fr &optional (base 'NL1))
loads the contents of **fr** into **gr**.

(rdstr gr &optional (base 'NL1))
loads the contents of the floating point status register into **gr**.

(wtfr gr fr &optional (base 'NL1))
writes the contents of **gr** to **fr**.

(wtstr gr &optional (base 'NL1))
writes the contents of **gr** to the floating point status register.

(cisl gr fr &optional (base 'NL1))
converts the 32 bit floating pointer number in **gr** to a 64 bit floating point number leaving the result in **fr** pair.

(cls fr1 fr2 &optional (base 'NL1))
converts the 64 bit floating point number in **fr1** pair to a 32 bit floating point number leaving the result in **fr2**. **Fr1** is not changed.

(cils gr fr1 fr2 &optional (base 'NL1))
writes the contents of **gr** which is the high half of the a 64 bit flonum to **fr1**, converts the **fr1** pair to a single float leaving the result in **fr2**.

(coms fr1 fr2 &optional (base 'NL1))
compares the single floats in **fr1 fr2** and sets the floating point condition codes appropriately. You need to read the floating point status register to get the results of the comparison.

(comis gr fr1 fr2 &optional (base 'NL1))
writes the contents of **gr** to **fr1** and compares this value with the contents of **fr2** setting the floating point condition codes appropriately.

(coml fr1 fr2 &optional (base 'NL1))
compares the two long floats in **fr1** and **fr2** pairs, setting the floating point condition codes appropriately.

(comil gr fr1 fr2 &optional (base 'NL1))
writes the contents of **gr** (which should be the high half of a 64 bit float) to **fr1** and compares the long floats in **fr1** and **fr2** pairs setting the floating point condition codes appropriately.

(fixnum-to-short gr fr &optional (base 'NL1))
converts the 32 bit integer in **gr** to a 32 bit floating point number leaving the result in **fr**.

(fixnum-to-long gr fr &optional (base 'NL1))
converts the 32 bit integer in gr to a 64 bit floating point number leaving the result in **fr** pair.

(abss fr1 fr2 &optional (base 'NL1))
takes the absolute value of **fr1** leaving the result in **fr2**.

(absl fr1 fr2 &optional (base 'NL1))
takes the absolute value of **fr1** pair leaving the result in **fr2** pair.

(adds fr1 fr2 &optional (base 'NL1))
The 32 bit float in **fr1** is added to the 32 bit float in **fr2** leaving the result in **fr2**.

(addis gr fr1 fr2 &optional (base 'NL1))
The 32 bit float in **gr** is written to **fr1**. **Fr1** is added to **fr2** leaving the result in **fr2**.

(addsi gr fr1 fr2 &optional (base 'NL1))
The 32 bit float in **gr** is written to **fr2**. **Fr1** is added to **fr2** leaving the result in **fr2**.

(addl fr1 fr2 &optional (base 'NL1))
The 64 bit float in **fr1** pair is added to the long float in **fr2** pair leaving the result in **fr2** pair.

(addil gr fr1 fr2 &optional (base 'NL1))
The high order 32 bits of the long float in **gr** is written to **fr1**. **Fr1** pair is added to **fr2** pair leaving the result in **fr2**.

(addli gr fr1 fr2 &optional (base 'NL1))
The high order 32 bits of the long float in **gr** is written to **fr2**. **Fr1** pair is added to **fr2** pair leaving the result in **fr2**.

(divs fr1 fr2 &optional (base 'NL1))
The 32 bit float in **fr2** is divided by **fr1** leaving the result in **fr2**.

(divis gr fr1 fr2 &optional (base 'NL1))
The 32 bit float in **gr** is written to **fr1**. **Fr2** is divided by **fr1** leaving the result in **fr2**.

(divsi gr fr1 fr2 &optional (base 'NL1))
The 32 bit float in **gr** is written to **fr2**. **Fr2** is divided by **fr1** leaving the result in **fr2**.

(divl fr1 fr2 &optional (base 'NL1))
The long float in **fr2** pair is divided by **fr1** pair leaving the result in **fr2** pair.

(divil gr fr1 fr2 &optional (base 'NL1))
The high order 32 bits of the long float in **gr** is written to **fr1**. **Fr2** pair is divided by **fr2** pair leaving the result in **fr2** pair.

(divli gr fr1 fr2 &optional (base 'NL1))
The high order 32 bits of the long float in **gr** is written to **fr2**. **Fr2** pair is divided by **fr1** pair leaving the result in **fr2** pair.

(muls fr1 fr2 &optional (base 'NL1))
The 32 bit float in **fr1** is multiplied by the 32 bit float in **fr2** leaving the result in **fr2**.

(mulis gr fr1 fr2 &optional (base 'NL1))
The 32 bit float in **gr** is written to **fr1**. **Fr1** is multiplied by **fr2** leaving the result in **fr2**.

(mulsi gr fr1 fr2 &optional (base 'NL1))
The 32 bit float in **gr** is written to **fr2**. **Fr1** is multiplied by **fr2** leaving the result in **fr2**.

(mull fr1 fr2 &optional (base 'NL1))
The 64 bit float in **fr1** pair is multiplied by the long float in **fr2** pair leaving the result in **fr2** pair.

(mulil gr fr1 fr2 &optional (base 'NL1))
The high order 32 bits of the long float in **gr** is written to **fr1**. **Fr1** pair is multiplied by **fr2** pair leaving the result in **fr2**.

(mulli gr fr1 fr2 &optional (base 'NL1))
The high order 32 bits of the long float in **gr** is written to **fr2**. **Fr1** pair is multiplied by **fr2** pair leaving the result in **fr2**.

(negs fr1 fr2 &optional (base 'NL1))
negates the value of **fr1** leaving the result in **fr2**.

(negl fr1 fr2 &optional (base 'NL1))
negates the value of **fr1** pair leaving the result in **fr2** pair.

(subs fr1 fr2 &optional (base 'NL1))
The 32 bit float in **fr1** is subtracted from **fr2** leaving the result in **fr2**.

(subis gr fr1 fr2 &optional (base 'NL1))
The 32 bit float in **gr** is written to **fr1**. **Fr1** is subtracted from **fr2** leaving the result in **fr2**.

(subsi gr fr1 fr2 &optional (base 'NL1))
The 32 bit float in **gr** is written to **fr2**. **Fr1** is subtracted from **fr2** leaving the result in **fr1**.

(subl fr1 fr2 &optional (base 'NL1))
The long float in **fr1** pair is subtracted from **fr2** pair leaving the result in **fr2** pair.

(subil gr fr1 fr2 &optional (base 'NL1))
The high order 32 bits of the long float in **gr** is written to **fr1**. **Fr1** pair is subtracted from **fr2** pair leaving the result in **fr2** pair.

(subli gr fr1 fr2 &optional (base 'NL1))
The high order 32 bits of the long float in **gr** is written to **fr2**. **Fr1** pair is subtracted from **fr2** pair leaving the result in **fr1** pair.

11.7. Useful Miscops

Most of the miscops should be called by the following sequence:

```
 (save-registers-pc non-pc-registers-to-be-saved)

load A0 with first argument.
load A1 with second argument.
load A2 with third argument.
load A3 with fourth argument.
store rest of the arguments on the stack.
(bali   PC miscop)

(restore-registers-pc non-pc-registers-saved-in-order)
```

Unless otherwise noted, the above calling sequence is the way to call a miscop from a miscop. Unless otherwise mentioned, miscops are free to destroy any of the first six registers (i.e., NL0, NL1, A0, A1, A2, and A3).

For some internal miscops, the arguments are passed in different registers. Also, rather than using the normal PC register, some use A3 for the return address. This convention is used to reduce the overhead of saving and restoring registers in some important cases.

The manual *Internal Design of Common Lisp on the IBM RT PC* describes many of the miscops that you can use. Under no circumstances, **repeat** never, use any of the allocation miscops. These miscops assume that they are being called from Lisp and may decide to see if it is time to invoke a garbage collection. This is done by escaping to Lisp code and thus a miscop will never regain control if a test for a GC is invoked.

multiply (X Y) multiplies the 32 bit number X by the 32 bit number Y. X is passed in NL0 and Y is passed in NL1. The return address is in A3. The high order result is returned in NL0 and the low order result is returned in NL1. This miscop does not modify any other registers. It does modify the MQ system control register and the condition codes.

divide (X Y) divides the 32 bit number X by the 32 bit number Y. X is passed in NL0 and Y in NL1. Y should not be -1, 0, or 1. These cases should be checked for before this miscop is called. The remainder is returned in NL0 and the quotient in NL1. This miscop modifies A3, the MQ system control register and the condition codes.

convert-bignum-to-single (x)
converts a bignum to a 32 bit flonum. The return address is in A3. The resulting 32 bit flonum

is returned in NL0.

convert-bignum-to-long (x)
converts a bignum to a 64 bit flonum. The return address is in A3. The high order part of the flonum is returned in NL0, the low order part in NL1.

convert-ratio-to-single (x)
converts a ratio to a 32 bit flonum. The return address is in A3. The resulting 32 bit flonum is returned in NL0.

convert-ratio-to-long (x)
converts a ratio to a 64 bit flonum. The return address is in A3. The high order part of the flonum is returned in NL0, the low order part in NL1.

11.8. Loading Miscops

Once a file containing miscops has been assembled, it can be loaded as follows:

```
(load "miscops.fasl")
(system:resolve-loaded-assembler-references)
```

The first line just loads in a file containing miscops. However, any external references made by the miscops will not be resolved. If you have several files with miscops that refer to one another, you should load in all the files, before doing (resolve-loaded-assembler-references). The call to resolve-loaded-assembler-references resolves all the external references of the files loaded up to the point that resolve-loaded-assembler-references is called.

11.9. Invoking User Miscops

Once a miscop has been loaded into a running Lisp, it is possible to call it. Assume you have loaded a miscop named foo into Lisp, you can call it by typing:

(extensions:call-user-miscop clc::foo Arg$_1$ ... Arg$_n$)

This will invoke the miscop with the arguments specified. Note that a miscop accepts a fixed number of arguments. You can not have optional or any other form of argument passing.

You can compile a function having a call to a user miscop. The compiler will generate the appropriate code to call the miscop. Any compiled files that reference a miscop must be loaded after the miscop has been loaded. If you don't do this, an error will be generated because the miscop will be undefined.

11.10. Tak Example

```
;;; Lisp code for TAK.

(defun tak (x y z)
  (declare (fixnum x y z))
  (if (not (< y x)) z
      (tak (tak (the fixnum (1- x)) y z)
           (tak (the fixnum (1- y)) z x)
           (tak (the fixnum (1- z)) x y))))
```

```
;;; Define a function that calls the tak-miscop.  This
;;; function should be compiled and loaded after the tak
;;; miscop has been loaded.  This must be in a separate
;;; file.

(defun call-tak-miscop (x y z)
  (extensions:call-user-miscop clc::tak x y z))
```

Following is miscop code for Tak, a simple benchmark. The code here is much better than that generated by the compiler.

```
(in-package 'compiler)

(define-user-miscop tak
  (save-registers-pc L0 L1 L2 L3 L4 BS FP AF)

  (bali PC tak-aux)               ; Go do real work.

  (restore-registers-pc L0 L1 L2 L3 L4 BS FP AF)
  (br   PC)                       ; Return to caller.
) ; end of tak.
(define-user-miscop tak-aux
  (lr   A3 A0)            ; Save X.
  (c    A3 A1)            ; Compare X and Y.

  (brlex PC)              ; Return to caller with result.
  (lr   A0 A2)            ; Move Z into return register.

  (cal  CS CS 24)         ; Enough room for 6 regs.
  (stm  R10 CS -20)       ; Save registers.
  (lr   R14 A3)           ; Save arg registers.
  (lr   R13 A1)
  (lr   R12 A2)

  (balix PC tak-aux)      ; Call tak-aux.
  (ai   A0 R14 -1)        ; First arg = X - 1.
                          ; Rest are set up.
  (lr   R11 A0)           ; Save result for later.

  (ai   A0 R13 -1)        ; First arg = Y - 1.
  (lr   A1 R12)           ; Second arg = Z.
  (balix PC tak-aux)      ; Call tak-aux.
  (lr   A2 R14)           ; Third arg = X.
  (noop)                  ; Padding for balix.
  (lr   R10 A0)           ; Save result for later.

  (ai   A0 R12 -1)        ; First arg = Z - 1.
  (lr   A1 R14)           ; Second arg = X.
  (balix PC tak-aux)      ; Call tak-aux.
  (lr   A2 R13)           ; Third arg = Y.
  (noop)

  (lr   A2 A0)            ; Get third arg.
  (lr   A0 R11)           ; Get saved value as first.
  (lr   A1 R10)           ; Get saved value as second.
  (lm   R10 CS -20)       ; Restore registers.
  (b    tak-aux)          ; Do tail recursive call.
)
```

Assume the tak miscop code is in the file takm.romp, the tak lisp code is in tak.lisp, then to execute the above code you could type:

```
    (clc:assemble-file "takm.romp")
    (load "takm.fasl")
    (system:resolve-loaded-assembler-references)
    (compile-file "tak.lisp")
    (load "tak.fasl")
    (call-tak-miscop 18 12 6)
```

In case you're wondering, the tak miscop runs in about 0.27 seconds of elapsed time compared to the compiled function time of 0.78.

Index

Index