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

David B. McDonald, *Editor*

April 1989

CMU-CS-89-132

School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

This is a revised version of Technical Report 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), ARPA Order No. 4976 under contract F33615-87-C-1499 and monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Aeronautical Systems Division (AFSC), Wright-Patterson AFB, OHIO 45433-6543.

Table of Contents

List of Tables

Acknowledgements

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

Chapter 1

Introduction

CMU Common Lisp is 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 a 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 or IBM RT PC/APC 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), IBM 6153 (APA8), or IBM 5080 (mpel) are supported) under the X window manager version 11 or a standard terminal, such as a Concept-100 or H-19.

At CMU, there is a misc collection named `cs.misc.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 several files including lisp, lisp.core, spelldict.bin, etc. 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 several files that it expects to find in /usr/misc/.lisp/lib/. If the inspector is used, then it expects to find some files in the directory /usr/misc/.lisp/lib/. The inspector expects to find a help file in the directory /usr/misc/.lisp/doc/.

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:

-core	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 specified core file is loaded.
-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, the name [machine:userid.processid]Editor is used. A file to edit may be specified by placing the name of the file between the program name (usually `lisp`) and the first 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.
-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.
-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.
-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.
-load	accepts an argument which should be the name of a file to load into Lisp before entering Lisp's read-eval-print loop.
-register	specifies that a slave Lisp should register with the nameserver. If an optional name is given, the slave Lisp registers that name with the nameserver. Otherwise, the default name [machine:userid]Eval is registered with the nameserver.
-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:userid.processid]Editor. If an optional name is given, the slave Lisp tries to connect to the named editor Lisp.

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 -core=mylisp.core
    lisp -core 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.

`extensions:*ignore-floating-point-underflow*` [*Variable*]

The variable **`*ignore-floating-point-underflow*`** controls what happens when floating point underflow occurs. If this variable is nil, a floating point underflow error is signalled. Otherwise, the operation causing the undeflow quietly returns a floating point zero of the appropriate type. The default value is **`nil`**.

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 *Common Lisp: the Language*. 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):

`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.
`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.
`conditions`	The **`conditions`** package contains the new error system as proposed for Common Lisp and exports several symbols necessary for the new error system.
`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.
`dired`	The **`dired`** package contains support functions for Hemlock's directory editing mode.
`edit`	The **`edit`** package contains matchmaker code to allow a Lisp running the eval server to establish a connection to a Lisp running Hemlock.
`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.
`inspect`	The **`inspect`** package contains the inspector.
`iterate`	The **`iterate`** package contains code used by CLOS and exports a few symbols needed by CLOS.

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).
lisp	The **lisp** package exports all the symbols defined by *Common Lisp: the Language* and only those symbols. Strictly portable Lisp code will depend only on the symbols exported from the **lisp** package.
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.
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.
netname	The **netname** package contains matchmaker code to interface to the name server. It exports the functions necessary to access the name server and some constants.
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.
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.
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.
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.
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.
walker	The **walker** package contains code used by CLOS and exports a few symbols needed by CLOS.
xlib	The **xlib** package contains the Common Lisp X interface (CLX) to the X11 interface. This is mostly Lisp code with a couple of functions that are defined in C to connect to the server.
xp	The **xp** package contains a version of Richard C. Waters' pretty printer written in Common Lisp.

The **lisp**, **user**, **keyword**, and **system** packages are required by *Common Lisp: the Language*.

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.

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 value of internal-time-units-per-second is 100.

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 value for the following variable, 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 a GC should run without much (if any) paging. GC's run more frequently but tend to take only about 5 seconds.

The following variables control the behavior of the garbage collector.

extensions:*bytes-consed-between-gcs* [*Variable*]

CMU Common Lisp automatically does a GC whenever the amount of memory allocated to dynamic objects exceeds the value of an internal variable. After each GC, this internal variable is set the amount of dynamic space in use at that point plus the value of the variable ***bytes-consed-between-gcs***. The default value is 2000000.

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. The default value is T.

extensions:*gc-notify-before* [*Variable*]

The variable **extensions:*gc-notify-before*** can be set to a function that should notify the user when a garbage collections begins. The function should accept one parameter: the amount of dynamic space in use before the GC. The default value of this variable is a function that prints a message similar to the following:

[GC threshold exceeded with 2,107,124 bytes in use. Commencing GC.]

extensions:*gc-notify-after* [*Variable*]

The variable **extensions:*gc-notify-after*** can be set to a function that should notify the user when a garbage collection finishes. This function should accept three parameters: the amount of dynamic spaced retained by the GC, the amount of dynamic space freed, and the new threshold which is the minimum amount of space in use before the next GC will occur. The default value of this variable is a function that prints a message similar to the following:

[GC completed with 25,680 bytes retained and 2,096,808 bytes freed.]
[GC will next occur when at least 2,025,680 bytes are in use.]

Note that a garbage collection will not happen at exactly the new threshold printed by the default ***gc-notify-after*** function. The system periodically checks whether this 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. However, this is not recommended.

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.

2.10. Load

An extension has been made to load to allow the user to control what happens when the object file is older than the corresponding source file.

extensions:*load-if-source-newer* *[Variable]*

The legal values for ***load-if-source-newer*** and their meanings are:

:load-object	The object file is loaded even though the source file is newer. This is the default.
:load-source	The source file is loaded instead of the older object file.
:compile	The source file is compiled and then the new object file is loaded.
:query	The user is asked a yes or no question to determine whether the source or object file is loaded.

If ***load-if-source-newer*** contains any other value, an error is signalled.

2.11. Modules

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

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.

2.12. The Inspector

An inspector that runs under the X window manager version 11 or on a tty is available in CMU Common Lisp.

inspect **&optional** *object* *[Function]*

Inspect calls the inspector on the optional argument *object*. If *object*, is not given, inspect immediately returns NIL. Otherwise, the behavior of inspect depends on whether Lisp is running under X or not.

If X is available, inspect creates an X window and displays *object* in the window. While inspect is running and the cursor is in the inspector's X window, mouse clicks and keyboard input have the following meaning:

Left	When the left mouse button is clicked over a component object, that object will be inspected in the current inspector window.
Middle	When the middle mouse button is clicked over a component object, inspect is exited returning the component as the result. All the new inspector windows are deleted.
Shift Middle	When the shift key is depressed and the middle mouse button is clicked over a component object, inspect exits and returns the component as the result. All the inspector windows are left displayed on the screen.
Right	When the right mouse button is clicked over a component object, that object will be inspected in a new inspector window.
d, D	When either d or D is typed, the current window is deleted. If there are no more windows, then inspect exits and returns the original *object*.
h, H, ?	When any of h, H, or ? are typed while in an inspector window, a new window with help information is displayed.
m, M	When either m or M is typed, a component object may be modified. The cursor changes to an arrow with an M beside it. Clicking any mouse button while the mouse is over a component will select that component as the destination for modification. If m was typed, the source object is also selected by the mouse which is indicated by an S beside the arrow in the cursor. If M was typed, the source object will be prompted for on the *query-io* stream. The source object replaces the destination object. While choosing the destination or source with the mouse, the operation can be aborted by type q or Q.
q, Q	When either q or Q is typed, inspect exits and returns the original *object*. All new inspector windows are deleted.
p, P	When either p or P is typed, inspect exits and returns the original *object*. All the . inspector windows are left on the screen.
r, R	When either r or R is typed, the current inspector display is recomputed. This is necessary to maintain a consistent display for an object that may have changed since the display was originally computed.
u, U	When either u or U is typed, the object of which the current object is a component is displayed. This is the inverse operation to clicking the left mouse button over a component object. If the window is currently displaying the top level object, nothing changes.

When the cursor is over a component object, the object is highlighted by surrounding it with a box.

If X is unavailable, a tty inspector is invoked. This inspector prints information bout and object and a numbered list of the components of the object. The following commands are available:

<n>	where <n> means a number corresponding to one of the components of the object. The inspector changes its focus to be this component. The inspector displays the components of the this new object.
r	recomputes the information for the current object.
d	redisplays the information for the current object.
u	moves up one level of the objects inspected. As you descend into the components of an object, a stack of all the objects previously seen is kept. This command pops you up one level of this stack.
q, e	quits the inspector returning the currently inspected object.
h, ?	displays some help text.

When inspect is eventually exited, it returns a Lisp object.

Chapter 3

Miscellaneous Extensions to Common Lisp

Several extensions have been made to make CMU Common Lisp a better development environment. This chapter describes various functions, macros, and variables beyond basic Common Lisp that have been added to CMU Common Lisp.

3.1. 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.

3.1.1. 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, SIGSEGV, and SIGFPE	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:

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.

3.1.2. 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
   (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.>
)
```

3.2. 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 -core file
```

3.3. Search Lists

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

$$\texttt{(setf (ext:search-list <name>) '(directory}_1\texttt{ directory}_2\texttt{ ...))}$$

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 (ext: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:

```
(ext: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 ext:search-list on code: as follows:

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

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

3.4. Running Programs from Lisp

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

`extensions:run-program` *program 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 pathname or string specifying the name of the file containing the program to run. *Args* should be a list of strings which are passed to *program* as normal Unix parameters. For no arguments, specify *args* as NIL. 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 value is :stream, then a stream is returned. Anything written to this stream is sent to the new process. The argument can also be a input stream that already contains all the input for the process. The input is sent to the process before run-program returns.

: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; **`:create`** creates the named file; and **`:error`** signals an error.

:output should be a pathname 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 value is :stream, then an input stream is returned which can be read from when the process has output. This stream is closed when eof is reached.

: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 run-program returns the process id (pid), input stream (if necessary), output stream (if necessary), and error stream (if necessary), plus the return values from the wait system call. When :wait is NIL, the same values are returned, except no values are returned for the wait system call (since the process will continue to run in the background).

3.5. Time Parsing and Formatting

Functions are provided to allow parsing strings containing time information and printing time in various formats are available.

`extensions:parse-time` *time-string* **`&key`** **`:error-on-mismatch :default-seconds`** **`:default-minutes :default-hours`** **`:default-day ...`** *[Function]*

`Parse-Time` accepts a string containing a time (e.g., "Jan 12, 1952") and returns the universal time if it is successful. If it is unsuccessful and the keyword argument *:error-on-mismatch* is non-NIL, it signals an error. Otherwise it returns NIL. The other keyword arguments have the following meaning:

:default-seconds	specifies the default value for the seconds value if one is not provided by *time-string*. The default value is 0.
:default-minutes	specifies the default value for the minutes value if one is not provided by *time-string*. The default value is 0.
:default-hours	specifies the default value for the hours value if one is not provided by *time-string*. The default value is 0.
:default-day	specifies the default value for the day value if one is not provided by *time-string*. The default value is the current day.
:default-month	specifies the default value for the month value if one is not provided by *time-string*. The default value is the current month.
:default-year	specifies the default value for the year value if one is not provided by *time-string*. The default value is the current year.
:default-zone	specifies the default value for the time zone value if one is not provided by *time-string*. The default value is the current time zone.
:default-weekday	specifies the default value for the day of the week if one is not provided by *time-string*. The default value is the current day of the week.

Any of the above keywords can be given the value *:current* which means to use the current value as determined by a call to the operating system.

`extensions:Format-Universal-Time` *dest universal-time* **`&key`** **`:timezone`** **`:style :date-first`** **`:print-seconds ...`** *[Function]*

`extensions:Format-Decoded-Time` *dest seconds minutes hours day month year* **`&key ...`** *[Function]*

`Format-Universal-Time` formats the time specified by *universal-time*. **`Format-Decoded-Time`** formats the time specified by *seconds*, *minutes*, *hours*, *day*, *month*, and *year*. *Dest* is any destination accepted by the **`format`** function. The keyword arguments have the following meaning:

:timezone	is an integer specifying the hours west of Greenwich. *:Timzone* defaults to the current time zone.
:style	specifies the style to use in formating the time. The legal values are:

:short	specifies to use a numeric date.
:long	specifies to format months and weekdays as words instead of numbers.
:abbreviated	is similar to long except the words are abbreviated.
:government	is similar to abbreviated, except the date is of the form "day month year" instead of "month day, year".

:date-first	if non-nil (default) will place the date first. Otherwise, the time is placed first.
:print-seconds	if non-nil (default) will format the seconds as part of the time. Otherwise, the seconds will be omitted.
:print-meridan	if non-nil (default) will format "AM" or "PM" as part of the time. Otherwise, the "AM" or "PM" will be omitted.
:print-timezone	if non-nil (default) will format the time zone as part of the time. Otherwise, the time zone will be omitted.
:print-seconds	if non-nil (default) will format the seconds as part of the time. Otherwise, the seconds will be omitted.
:print-weekday	if non-nil (default) will format the weekday as part of date. Otherwise, the weekday will be omitted.

Chapter 4

Error System

Written by Kent M. Pitman and Bill Chiles

4.1. Introduction

This chapter describes the Common Lisp Condition System, as proposed and accept by the X3J13 subcommittee on error handling. The design is primarily fixed, but the standards committee is still making changes to complete and polish it. Most of the work that remains is fully specifying the standard condition types (described below) and which Common Lisp functions must signal what conditions under what situations. CMU Common Lisp defines the conditions specified near the end of this chapter, but it does not signal all of them when you might expect. Therefore, we support a somewhat unsophisticated environment for extremely clever condition handling; however, you will probably find more functionality and conditions implemented than you'll need.

4.1.1. Purpose

Often we find it useful to describe a function in terms of its behavior in *normal situations*. For example, we may say informally that the function + returns the sum of its arguments or that the function **read-char** returns the next available character on a given input stream. Sometimes *exceptional situations* arise which do not fit neatly into such descriptions. For example, + might receive an argument which is not a number, or **read-char** might receive a single argument which was a stream that had no more available characters. This distinction between normal and exceptional situations is in some sense arbitrary, but is often very useful in practice.

For example, suppose you had a function **F** which you defined to allow only integer arguments, but you also guaranteed that the function **F** would detect and signal an error for non-integer arguments. Such a description is in fact internally inconsistent because the behavior is well-defined for non-integers. Yet we would not want this to force us to have to describe **F** as a function that accepts any kind of argument (just in case someone calls **F** only as a quick way to signal an error, for example). Using our new terminology, we can say clearly that **F** accepts integers in the normal situation, and signals an error in exceptional situations. Moreover, we can say that when we refer to the definition of a function informally, it is acceptable to speak only of its normal behavior. For example, we can speak informally about **F** as a function that accepts only integers without feeling that we are committing some fraud.

Not all exceptional situations are errors. For example, a program which is typing out a long line of text may notice that it is at the end of the line. It is possible that no real harm will result from continuing to type past the end of the line because the operating system will simply force a carriage return on the output device and continue typing on the next line. Even though the system recovers, it may be interesting to establish a protocol whereby that program can inform its callers of end-of-line exceptions. The controlling program could then opt to deal with these situations in interesting ways at certain times. It might choose to terminate printing, obtaining an end-of-line truncation. The point is the printer program can continue to operate correctly even when the controlling program fails to provide advice about the situation; the situation is not an error.

Mechanisms for dealing with exceptional situations vary widely. When one occurs, a program may attempt to handle the situation by returning a distinguished value, returning an additional value, setting a variable, calling a function, performing a special transfer of control, or by stopping the program altogether and entering the debugger. For the most part, the facilities described in this document do not introduce any fundamentally new way of dealing with exceptional situations; rather, they encapsulate and formalize useful patterns of data and control flow which programmers have found useful in dealing with exceptional situations.

A proper conceptual approach to errors should begin with a discussion of the principles of *conditions* in general and eventually work its way up to the concept of an *error* as just one of the many kinds of conditions. However, given the widespread primitive state of error handling technology, a proper buildup may be as inappropriate as requiring that a beggar learn to cook a gourmet meal before being allowed to eat. As such, this chapter will first deal with the essentials, error handling, and then later go back and fill in the missing details.

4.1.2. Terminology

Common Lisp: the Language (pp. 5-6) says the following about errors:

> When this manual specifies that it "is an error" for some situation to occur, this means that:
>
> - No valid Common Lisp program should cause this situation to occur.
> - If the situation occurs, the effects are completely undefined as far as adherence to the Common Lisp specification is concerned.
> - No Common Lisp implementation is required to detect such an error. Of course, implementors are encouraged to provide for detection of such errors wherever reasonable.
>
> This is not to say that some particular implementation might not define the effects and results for such a situation; the point is that no program conforming to the Common Lisp specification may correctly depend on such effects or results.
>
> On the other hand, if it is specified in this manual that in some situation "an error is *signalled*", this means that:
>
> - If this situation occurs, an error will be signalled (see **error** and **cerror**).
> - Valid Common Lisp programs may rely on the fact that an error will be signalled.
> - Every Common Lisp implementation is required to detect such an error.
>
> In places where it is stated that so-and-so "must" or "must not" or "may not" be the case, then it "is an error" if the stated requirement is not met. For example, if an argument "must be a symbol", then it "is an error" if the argument is not a symbol. In all cases where an error is to be *signalled*, the word "signalled" is always used explicitly in this manual.

This has changed for the language standard. We have some new terms and phrases for describing what built in functions do. A *condition* is an interesting situation in a program which has been detected and announced. Later we will allow this term to also refer to objects which programs use to represent such situations. An *error* is a condition in which normal program execution may not continue without some form of intervention, either interactively by the user or under some sort of program control as described later in this document. *Signalling* is the process by which a program formally announces a condition. The **signal** function is the primitive mechanism that makes such announcements. Other abstractions, such as **error** and **cerror**, are built using **signal**.

Common Lisp: the Language is ambiguous about the reason why a particular program action is an error. There are two principal reasons why an action may be an error without requiring the signalling of an error:

- Detecting the error might be prohibitively expensive. Consider the following example:

  ```
  (+ nil 3)
  ```

  This is an error. It is likely that the designers of Common Lisp believed that this would be an error in all implementations but that they felt that it might be excessively expensive to detect the problem in compiled code on stock hardware, so they did not require that + signal an error.

- Some implementations might implement the behavior as an extension. Consider the following example:

  ```
  (loop for x from 1 to 3 do (print x))
  ```

  This is an error because `loop` is not defined to take atoms in its body. Some implementations offer an extension which makes this well-defined. In order to leave room for such extensions, *Common Lisp: the Language* uses the ''is an error'' terminology to keep implementors from being forced to signal an error in the extended implementations.

This chapter uses the following terminology, which has been accepted by the X3J13's error handling subcommittee to become the standard in the next edition of the Common Lisp specification:

- If the signalling of a condition or error is part of a function's contract for specified situations, this documentation will say that it ''signals'' or ''must signal'' that condition or error.
- If the signalling of a condition or error is optional for some important reason, such as performance, this documentation will say that the program ''might signal'' that condition or error. In this case, it defines the operation to be illegal in all implementations, but allowing some implementations to avoid actually detecting the error.
- If an action is left undefined for the sake of an implementation-dependent extension, this documentation will say that it ''is undefined'' or ''has undefined effect.'' This means that it is not possible to portably depend upon the effects of that action. A program which has an undefined effect could do anything including entering the debugger, transfering control, or modifying data in unpredictable ways.
- In the special case where only the return value of an operation is undefined, but any side-effect and transfer-of-control behavior is well defined, this documentation will say that it has ''undefined value.'' In this case, the number and nature of the return values is undefined, but the user can reasonably expect the function to return. Under this description, there are some though not many, legitimate ways in which such return values can be used. For example, if the function `foo` has no side-effects and undefined value, the expression `(list (foo))` is completely well-defined even for portable code, but the effect of `(print (foo))` is not well-defined.

4.2. Concepts

4.2.1. Signalling Errors

Signalling an error in a program is an admission by that program that it does not know how to continue and requires external intervention. Once it signals an error, any decision about how to continue must come from outside of it.

The simplest way to signal an error is to use the **`error`** function with **`format`**-style arguments describing the error. If a piece of code calls **`error`**, and there are no active handlers (described later), the system enters the debugger and outputs the error message. For example, you might see an interaction such as the following:

```
* (defun factorial (x)
    (cond ((or (not (typep x 'integer)) (minusp x))
           (error "~S is not a valid argument to FACTORIAL." x))
          ((zerop x) 1)
          (t (* x (factorial (1- x))))))
FACTORIAL
* (factorial 20)
2432902008176640000
* (factorial -1)

Error in function FACTORIAL.
-1 is not a valid argument to FACTORIAL.

Restarts:
  0: Return to Top-Level.

Debug  (type H for help)
(FACTORIAL -1)
0]
```

A call to **error** cannot directly return. Unless a program prepares for special flow of control to override this behavior, **error** enters the debugger, and there will be no option to continue. An implementation's debugger may provide commands for interactively returning from individual stack frames, but the user must know what he is doing. The point is programs are written as if **error** never returns even though an implementation's environment may provide special development features.

A programmer may have a single, well-defined idea of a recovery strategy for an error, in which case he can use the function **cerror**. This specifies information to the user about what would happen if the user does continue from the call to **cerror**. For example:

```
* (defun factorial (x)
    (cond ((not (integerp x))
           (error "~s is not a valid argument to FACTORIAL." x))
          ((minusp x)
           (let ((x-magnitude (- x)))
             (cerror "Compute -(~D!) instead."
                     "(-~D)! is not defined."
                     x-magnitude)
             (- (factorial x-magnitude))))
          ((zerop x) 1)
          (t (* x (factorial (- x 1))))))
FACTORIAL
* (factorial -3)

Error in function FACTORIAL.
(-3)! is not defined.

Restarts:
  0: Compute -(3!) instead.
  1: Return to Top-Level.

Debug  (type H for help)
(FACTORIAL -1)
0] restart 0
-6
```

4.2.2. Trapping Errors

By default, **error** enters the debugger. You can override this behavior in a variety of ways; the simplest and most general mechanism is to wrap your code in an **ignore-errors** form. Normally forms in the body of **ignore-errors** evaluate sequentially returning the last value. If the evaluation of this code results in the signalling of a condition of type **error**, **ignore-errors** immediately returns two values: **nil** and the signalled condition object (described later). The system does not invoke the debugger or print any error messages. For example:

```
* (setq filename "nosuchfile")
"nosuchfile"
* (ignore-errors (open filename :direction :input))
NIL
#<FILE-ERROR.5EA4>
```

Usually, **ignore-errors** is undesirable because it handles every possible kind of error. Though some may argue differently, a program which avoids entering the debugger is not necessarily better than one which does enter it. Excessive use of **ignore-errors** keeps the user out of the debugger, but it does little to increase your program's reliability; your program may continue running after encountering errors other than those you designed it to ignore. In general, it is better to deal with the particular errors that you believe could occur, and if an unexpected error does happen, you will find out about it.

The error system defines **ignore-errors** on a more general facility called **handler-case**. It allows the user to specifically deal with types of conditions, including non-**error** conditions, without affecting the signalling of disjoint or more general kinds of conditions. The following example achieves an equivalent effect to the previous example's use of **ignore-errors**:

```
* (setq filename "nosuchfile")
"nosuchfile"
* (handler-case (open filename :direction :input)
    (error (condition) (values nil condition)))
NIL
#<FILE-ERROR.5EA9>
```

The advantage of **handler-case** in this scenario is the ability to specify a more specific condition type than **error**. Condition types are explained in detail later, but the following should be a clear example:

```
* (makunbound 'filename)
FILENAME
* (handler-case (open filename :direction :input)
    (file-error (condition) (values nil condition)))

Error: The variable FILENAME is unbound.

Restarts:
  1: Retry getting the value of FILENAME.
  2: Specify a value of FILENAME to use this time.
  3: Specify a value of FILENAME to store and use.
  4: Return to Top-Level.

Debug  (type H for help)
0]
```

4.2.3. Handling Conditions

The basic idea of condition handling involves the signalling of a condition. A piece of code called the *signaller* recognizes and announces an exceptional situation using **signal**, or some function built on it such as **error**. The process of signalling includes the search for and invocation of a *handler*, a piece of code that will attempt to take care of the situation appropriately. If this process finds a handler, it may either *handle* the situation by performing some non-local transfer of control, or it may *decline* by refusing to perform a non-local transfer of control. Whenever a handler declines, the search for a willing handler continues.

Since the lexical environment of the signaller might not be available to handlers, the system supports a data structure called a *condition* to represent the relevant state of the situation. Users can also create conditions explicitly using **make-condition** and pass them to a function such as **signal**, or they can allow the system to create conditions implicitly by using functions such as **signal** and **error**. To handle a condition a handler can use any non-local transfer of control including the following:

- **go** to a tag in a **tagbody**
- **return** from a **block**
- **throw** to a **catch**

The system provides abstractions built on these primitives for convenience in exception handling. For example, **handler-bind** makes a handler dynamically accessible to a program, and the following creates a handler for a condition of type **arithmetic-error**:

```
(handler-bind ((arithmetic-error #'this-handler))
  ...body...)
```

A handler is a function of one argument, a condition. While **body** executes, if someone signals a condition of the designated type, and there are no dynamically intervening handlers, **signal** invokes the handler on the given condition. The following is a complete example showing a macro that handles **arithmetic-error**'s by returning **nil** and the condition if the arithmetic could not be computed:

```
(defmacro without-arithmetic-errors (&body forms)
  (let ((tag (gensym)))
    `(block ,tag
       (handler-bind ((arithmetic-error
                       #'(lambda (condition)
                           (return-from ,tag
                                        (values nil condition)))))
         ,&body))))
```

Handlers execute in the dynamic context of the signaller, but the system rebinds the set of available condition handlers to those that were active at the time the program established the handler. This means that if the handler signals a condition or calls something that signals one, the handler and any others bound in the same **handler-bind** form are inaccessible to the signalling process.

If the system can only find handlers that decline, and the condition is signalled via **error** or **cerror**, or similar routines, the system enters the debugger within the dynamic context of the signaller.

4.2.4. Object-Oriented Basis of Condition Handling

The ability of the handler to usefully handle an exceptional situation is related to the quality of the information given to it. If we only signalled errors with a string describing the condition, **string-equal** would be a handler's best tool for identifying what happened, and the information presented to the user would be the same as the string passed to the handler. It would be ridiculous to try to map what was passed to the error system to something different to display to the user.

It is fundamentally important to decouple the error message string from the objects which formally represent the error state. Thus, there is a notion of typed conditions and formal operations on them which make them inspectable in a structured way. This object-oriented approach to condition handling has the following important advantages over a text-based approach:

- Conditions are classified according to subtype relationships, making it easy to test for categories of conditions.
- Conditions have named slot values through which parameters are conveyed from the program that signals the condition to the program that handles it.
- Inheritance of methods (in a loose sense) and slots reduce the amount of explicit specification necessary to achieve various interesting effects.

This document describes some predefined condition types, and the set of condition types is extensible using **define-condition**. The following is an example defining a function of two arguements called **divide** that is patterned after the / function and that does some error checking:

```
(defun divide (numerator denominator)
  (cond ((or (not (numberp numerator)) (not (numberp denominator)))
         (error "(DIVIDE '~S '~S) - Bad arguments."
                numerator denominator))
        ((zerop denominator)
         (error 'division-by-zero
                :operator 'divide
                :operands (list numerator denominator)))
        (t ...)))
```

In the first clause, the definition uses **error** with a string argument, and in the second clause it names a particular condition type, **division-by-zero**. In the case of a string argument, the system signals a **simple-error** condition type.

The particular kind of error signalled may be important when handlers are actually active. For example, **simple-error** inherits from type **error**, which in turn inherits from type **condition**. In the other case, **division-by-zero** inherits from **arithmetic-error**, which inherits from **error**, etc. If a handler existed for **arithmetic-error** when some code signals a **division-by-zero** condition, the system would invoke that handler; however, if the same code in the same context signals a **simple-error** condition, the system would ignore the handler for the **arithmetic-error** type.

4.2.5. Restarts

The condition system separates the act of signalling an error of a particular type from the means of recovering from that error in some way. In the **divide** example in the previous section, signalling an error does not imply a willingness on the part of the signaller to cooperate in any corrective action. For example, if the user ends up in the debugger, his only option may be to return to the Lisp top level.

When a program detects an error and calls **error**, execution cannot continue normally because **error** never returns directly. However, the user can write his program to transfer control to other points in the program with specially established *restarts*. The simplest restart involves structured transfer of control using a macro called **restart-case**. The **restart-case** form allows the programmer to execute a piece of code in a context where zero or more restarts are active, and if the program or the user, through the debugger, invokes one of those restarts, the system transfers control to the corresponding clause in the **restart-case** form.

The following shows the **divide** example from the previous section rewritten:

```
(defun divide (numerator denominator)
  (loop
    (restart-case (return
                    (cond ((or (not (numberp numerator))
                               (not (numberp denominator)))
                           (error "(DIVIDE '~S '~S) - Bad arguments."
                                  numerator denominator))
                          ((zerop denominator)
                           (error 'division-by-zero
                                  :operator 'divide
                                  :operands
                                  (list numerator denominator)))
                          (t ...)))
      (nil (arg1 arg2)
           :report
           "Provide new arguments for use by the DIVIDE function."
           :interactive (lambda ()
                          (list (prompt-for 'number "Numerator: ")
                                (prompt-for 'number "Denominator: ")))
           (setq numerator arg1 denominator arg2))
      (nil (result)
           :report
           "Provide a value to return from the DIVIDE function."
           :interactive
           (lambda () (list (prompt-for 'number "Result: ")))
           (return result)))))
```

The `nil` at the head of each clause means that it is an *anonymous* restart. Anonymous restarts are typically only invoked from within the debugger. Later sections describe in detail *named* restarts that programs can call, typically from handlers, without the need for user intervention. If the arguments to anonymous restarts are required, not optional, the code must specify the `:interactive` keyword to provide information concerning how to supply the arguments in case the user causes its invocation via the debugger.

The `:report` keyword specifies how to present the restart option to the user, such as in the debugger.

In this example, `prompt-for` is immaterial and does what you think.

The following is a sample interaction that takes advantage of the restarts provided by the revised definition of `divide`:

```
* (+ (divide 3 0) 7)

Error in function DIVIDE.
Attempt to divide 3 by 0.

Restarts:
  0: Provide new arguments for use by the DIVIDE function.
  1: Provide a value to return from the DIVIDE function.
  2: Return to Top-Level.

Debug  (type H for help)
(DIVIDE 3 0)
0] restart 0
Numerator: 4
Denominator: 2
9
```

4.2.6. Named Restarts

Named restarts are more powerful or convenient than unnamed ones since programs and users can invoke them without the aid of an interface like the debugger. The following is a degenerate, interesting example:

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

This adds 3 to 1, returning 4, and it is analagous to writing:

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

A more practical example below shows a possible portion of Lisp's **symbol-value** function that signals an **unbound-variable** error:

```
(restart-case (error 'unbound-variable :name variable)
  (continue ()
    :report (lambda (stream)
              (format stream "Retry getting the value of ~S."
                      variable))
    (symbol-value variable))
  (use-value (value)
    :report (lambda (stream)
              (format stream "Specify a value of ~S to use this time."
                      variable))
    value)
  (store-value (value)
    :report (lambda (stream)
              (format stream "Specify a value of ~S to store and use."
                      variable))
    (setf (symbol-value variable) value)
    value))
```

With this, users can write a variety of automatic handlers for **unbound-variable** errors. The following makes unbound variables evaluate to themselves:

```
(handler-bind ((unbound-variable
                #'(lambda (condition)
                    (if (find-restart 'use-value)
                        (invoke-restart
                         'use-value
                         (cell-error-name condition))))))
  ...body...)
```

4.2.7. Restart Functions

Some restarts or recovery techniques are common in that programmers find themselves writing very similar restart cases. It is good style to provide a simpler invocation means than is otherwise used for these. *Restart functions* hide the typical use of **invoke-restart**.

Conventionally the restart function shares the name of the restart name. The system defined functions **abort**, **continue**, **muffle-warning**, **store-value**, and **use-value** are restart functions. With **use-value**, the **handler-bind** example at the end of the previous section that handles **unbound-variable** errors becomes much simpler:

```
(handler-bind ((unbound-variable
                #'(lambda (condition)
                    (use-value (cell-error-name condition)))))
  ...body...)
```

Textually the example only saves two lines of code, but conceptually the handler doesn't have to be concerned with whether the restart is currently active. You don't want your handler to get an unactive restart error because you forgot to make sure the restart exists. **Use-value** takes care of that and simply returns if the restart is unactive, causing the handler to return indicating that it declines handling the condition.

4.2.8. Contrasting Restarts and Catch/Throw

One important feature **restart-case** offers which **catch/throw** does not is the ability to reason about the available points to which a program transfers control without actually attempting the transfer. Considering the following, the first form is a poor man's variation of the second:

```
(ignore-errors (throw ...))

(if (find-restart 'something) (invoke-restart 'something))
```

The following two forms are much cleaner than the programming required using **ignore-errors**, **throw**, and hacks with binding specials to know what context you are in:

```
(if (and (find-restart 'something) (find-restart 'something-else))
    (invoke-restart 'something))

(if (and (find-restart 'something) (yes-or-no-p "Do something? "))
    (invoke-restart 'something))
```

Simply using **ignore-errors** and **throw** forces a transfer of control when it possibly is inconvenient or an error, and the restart mechanism readily provides a means for inspecting the dynamic context and interacting with the user.

Another difference between the restart facility and the **catch/throw** facility is that a **catch** with any given tag completely shadows any outer, pending **catch** with the same tag. Because of the **compute-restarts** function, it is possible to see shadowed restarts which can be very useful, such as in the debugger.

4.2.9. Generalized Restarts

Restart-case allows only imperative transfer of control for its associated restarts. The system defines it using a lower level primitive called **restart-bind** which does not force transfer of control. Its syntax is as follows:

```
(restart-bind ((name function . options)) . body)
```

body executes in a dynamic context where **(invoke-restart 'name)** invokes **function**. The **options** are keyword-style and describe information similar to that provided with the **:report** keyword in **restart-case**. A **restart-case** expands into a call to **restart-bind** with functions that unconditionally transfer control to a particular body of code, passing along arguments.

Restarts can be useful without transfering control. Consider the following example:

```
(restart-bind ((nil #'(lambda () (expunge-directory the-dir))
                    :report-function
                    #'(lambda (stream)
                        (format stream "Expunge ~A."
                                (directory-namestring the-dir)))))
   (cerror "Try this file operation again."
           'directory-full :directory the-dir))
```

Entering the debugger in this context, the user could perform the expunge, avoiding transfering control from within the debug context, and then retry the file operation, as in:

```
* (open "foo" :direction :output)

Error in function OPEN.
The directory /usr/bovik/ is full.

Restarts
  0: Try this file operation again.
  1: Expunge /usr/bovik/.
  2: Return to Lisp Top-Level.

Debug  (type H for help)
(OPEN "foo" :DIRECTION :OUTPUT)
0] restart 1
Expunging /usr/bovik/ ...
0] restart 0
#<File stream "/usr/bovik/foo">
```

4.2.10. Serious Conditions

The `ignore-errors` macro will trap conditions of type `error`, but since this form is so dangerous for squelching every kind of error, some conditions are very serious without being a subtype of `error`. These are of type `serious-condition`, and the system might use this type for situations such as stack overflow or exhausted storage. The type `error` is a subtype of `serious-condition`, and though it is technically correct to refer to errors as *serious conditions*, we typically reserve that phrase to indicate conditions that are subtypes of `serious-condition` excluding subtypes of `error`.

This distinction is necessary to provide for exceptions that don't fall under the domain of the Common Lisp language. We assume an implementation uses a stack for function calling, and we know that stacks can overflow; however, this is not a programming error. In another implementation, the same program might run fine. Furthermore, if a program is dynamically within an `ignore-errors` form, and the system runs out of memory or the stack overflows, the system must stop or take care of this. The conditions simply cannot be ignored.

By convention, programmers prefer the function `error` over `signal` to signal conditions of type `serious-condition`, as well as those of type `error`. It is the use of the function `error`, and not the type of the signalled condition, that causes the system to enter the debugger.

4.2.11. Non-Serious Conditions

Some conditions are neither errors nor serious conditions. Programs signal these to give other programs a chance to intervene, but if none take any action, then computation simply continues normally. For example, an implementation might choose to signal a non-serious condition called `end-of-line` when output reaches the last character position on a line of character output. In such an implementation, the signalling of this condition allows a convenient way for other programs to do something special in this situation, producing output that is truncated at the end of a line or simulating a line-wrapping device.

Use `signal` to signal these types of conditions. If the program uses `error` to signal a non-serious condition, the system will still enter the debugger if it goes unhandled. The point of signalling a non-serious condition is that it should not matter if a program continues to execute immediately after the signalling regardless of whether some other program took any action based on the situation.

4.2.12. Condition Types

Some types of conditions are predefined by the system. All types of conditions are subtypes of `condition`. That is, `(typep c 'condition)` is true if and only if `c` is a `condition` object.

Implementations supporting multiple (or non-hierarchical) type inheritance are expressly permitted to exploit multiple inheritance in the tree of condition types as implementation-dependent extensions, as long as such extensions are compatible with this document.

In order to avoid problems in portable code which run both in systems with multiple type inheritance and systems without it, the designers warn against assuming subtype relationships specified in this document are mutually exclusive. In some cases this document does specify disjoint subtypes, but this is not the default. For example, from the subtype descriptions contained in this document, in all implementations the following must be true:

```
    (typep c 'control-error) implies (typep c 'error),
```

However, the reader must avoid the following assumption:

```
    (typep c 'control-error) implies (not (typep c 'cell-error))
```

4.2.13. Signalling Conditions

When a program signals a condition, the system tries to locate the most appropriate handler for the condition and invoke that handler. There are constructs for dynamically establishing handlers. If the process of signalling finds a suitable handler, it calls the handler. Sometimes handlers decline by simply returning without performing a non-local transfer of control. When this happens, the search for an appropriate handler continues as if the handler never existed. When `signal` fails to find a handler to take care of the situation, it returns `nil`.

It is worth noting the handler search procedure finds dynamically more local handlers before it finds those established dynamically earlier in time, regardless of whether the more local handler is more specific than any earlier bound handler. Therefore, the programmer should take care when binding handlers to very general condition types since a more specific handler may already be established that is more appropriate. There is no reason to be overly concerned about this, experience with existing condition systems suggests that this is a reasonable approach and works adequately in most situations.

4.2.14. Condition Handlers

A *handler* is a function of one argument, the signalled condition. The handler may inspect the object to see if it really wants to take care of it. Handlers execute in the dynamic context of the signaller, but the system rebinds the set of available condition handlers to those that were active at the time the program established the handler. The intent of this is to prevent infinite recursion due to errors in a condition handler.

After inspecting the condition, the handler should take one of the following actions:

- *Decline* to handle the condition by simply returning.
- Handle the condition by performing some non-local transfer of control. This may be done either primitively using `go`, `return`, and `throw` or more abstractly using a function such as `abort` or `invoke-restart`.
- Signal another condition.
- Invoke the interactive debugger.

The latter two items are really ways of putting off the decision to either handle or decline, in case some other code or the user wants to get in on the recovery action. Ultimately, all a handler can do is handle or decline to handle a condition.

4.2.15. Printing Conditions

When `*print-escape*` is `nil`, as with `princ` or the `~A format` option, the system invokes the report method for the condition. This is the means for presenting conditions to users. Some functions, **`invoke-debugger`**, **`break`**, and **`warn`**, always display the condtion, but users can explicity cause conditions to report themselves when desired:

```
(defun open-data-file (user-specified-name default-system-name)
  (handler-case (open user-specified-name)
    (serious-condition (condition)
      (format t "~&Opening ~S failed:~%~A~&Using ~S instead."
              user-specified-name condition default-system-name)
      (open default-system-name))))
```

This might print something like the following:

```
Opening #.(pathname "/usr/dat/unavailable-data-file") failed:
#.(pathname "/usr/dat/unavailable-data-file") is read protected.
Using #.(pathname "/usr/dat/default-objects") instead.
```

Some notes about the text presented by report methods:

- The message should be a complete sentence, beginning with a capital letter and ending with appropriate punctuation.
- The message should exclude introductory text such as "`Error:`" or "`Warning:`". Such text will be added by the routine invoking the report method as appropriate to the context.
- Except where unavoidable, tab characters should be avoided in error messages. Their effect can vary between implementations and can cause problems even within an implementation because it may output a variety of space depending on the current printing column when the condition reports.
- Single line messages are preferred, but newlines in the middle of long messages are acceptable.
- If any program displays messages indented from the prevailing left margin, possibly to make the report standout against other text, then that program will take care to insert the indentation into any extra lines of a multi-line error message. Similarly, any program that prefixes error messages with semicolons so that they appear to be comments should take care of inserting a semicolon at the beginning of each line in a multi-line error message.

When `*print-escape*` is non-`nil`, the object should print in some useful and fairly abbreviated fashion according to the style of the implementation. The condition may print unreadably, as by **`read`**; that is, it may use "#<" syntax.

4.3. Signalling Conditions

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

This function invokes the signal facility on a condition formed from *datum* and *arguments*. If the condition is unhandled, this calls **`invoke-debugger`** on the condition. This function never returns and can only be exited by a non-local transfer of control in a handler or by use of a debugger command.

This uses *datum* and *arguments* as follows:

- If *datum* is a condition, then this uses it directly. In this case, it is an error for *arguments* to be anything other than `nil`.
- If *datum* is a condition type, then this uses the condition resulting from `apply`'ing `make-condition` to *datum* and *arguments*.
- If *datum* is a string, then this uses the condition resulting from the following:

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

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

This function invokes the error facility on a condition formed from *datum* and *arguments*. If the condition is unhandled, this calls **`invoke-debugger`** on the condition. While signalling the condition, and while in the debugger if it is reached, it is possible to continue program execution using the **`continue`** restart.

This uses *datum* and *arguments* as follows:

- If *datum* is a condition, then this uses it directly. In this case, **`cerror`** only uses *arguments* with *continue-format-string*, and it will not use *arguments* to initialize *datum* in any way.
- If *datum* is a condition type, then this uses the condition resulting from **`apply`**'ing **`make-condition`** to *datum* and *arguments*. In this case, **`cerror`** uses *arguments* with *continue-format-string* in a call to **`format`** and with *datum* in a call to **`make-condition`**, so the user must take care to set up the **`format`** string correctly. The directive **`~*`** may be useful in this situation.
- If *datum* is a string, then this uses the condition resulting from the following:

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

continue-format-string must be a string, and **`cerror`** returns **`nil`**.

`signal` *datum* **`&rest`** *arguments* [*Function*]
`*break-on-signals*` [*Variable*]

This function invokes the signal facility on a condition formed from *datum* and *arguments*. If the condition is unhandled, **`signal`** returns **`nil`**.

This uses *datum* and *arguments* as follows:

- If *datum* is a condition, then this uses it directly. In this case, it is an error for *arguments* to be anything other than **`nil`**.
- If *datum* is a condition type, then this uses the condition resulting from **`apply`**'ing **`make-condition`** to *datum* and *arguments*.
- If *datum* is a string, then this uses the condition resulting from the following:

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

If the following test is true, then this function enters the debugger before beginning the signalling process:

```
(typep condition *break-on-signals*)
```

The user can continue this invocation of the debugger using the **`continue`** restart. Note, this is true for functions and macros that use **`signal`**: **`error`**, **`cerror`**, **`warn`**, **`assert`**, and **`check-type`**.

The condition system provides **`*break-on-signals*`** for debugging programs that do signalling. The user should choose the most restrictive specification that suffices. Setting this flag effectively violates the modular handling of condition signalling this system seeks to establish, and the effect may be unpredictable in some cases since the user may not be aware of the variety or number of calls to **`signal`** large sophisticated programs use.

warn *datum* **&rest** *arguments* [*Function*]
break-on-warnings [*Variable*]

This function warns about a situation by signalling a condition of type **warning** formed from *datum* and *arguments*.

This uses *datum* and *arguments* as follows:

- If *datum* is a condition, this uses condition directly. In this case, if the condition is not of type **warning**, or *arguments* is something other than **nil**, **warn** signals an error of type **type-error**.
- If *datum* is a condition type, then this uses the condition resulting from **apply**'ing **make-condition** to *datum* and *arguments*. If this is anything other than a subtype of **warning**, **warn** signals a **type-error** error.
- If *datum* is a string, then this uses the condition resulting from the following:

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

If ***break-on-warnings*** is true, then the **warn** enters the debugger using **break** before signalling the **warning** condition. Because of the use of **break**, the **continue** restart allows **warn** to continue executing normally. This feature is only supported for compatibility with previous condition system proposals that some implementations implemented. The ***break-on-signals*** mechanism supersedes ***break-on-warnings*** and is more general in comparison. Programmers should write new code using the ***break-on-signals***, and if they want to break on warnings, then they should set the variable to **'warning**. The condition system provides these features for debugging programs that issue warnings.

The precise mechanism for warning is as follows:

1. If ***break-on-warnings*** is true, **warn** calls **break**. If the user continues the break the **continue** restart, proceed with step 2.
2. Signal the **warning** condition with an active **muffle-warning** established. This allows handlers to cause **warn** to immediately return **nil** without taking any other action. If the condition goes unhandled, proceed with step 3.
3. Report the condition to ***error-output***.
4. Return **nil**.

4.4. Handling Conditions

handler-case *form* {*case*}* [*Macro*]

This macro executes *form* in a context where various handlers are active as specified by each *case*. Each *case* is of the following form:

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

Type may be any type specifier. If during the execution of *form*, the code signals a condition for which there is an appropriate clause (that is, the condition's type is a subtype of one of the specified types), and there is no intervening handler for the condition's type, then the system transfers control to the body of the clause. The code in the clause executes in the dynamic context of the signaller, but with respect to bound handlers, the system alters the context to that immediately prior to establishing the invoked handler. The code executes also with *var* bound to the signalled condition. If *form* runs to a normal completion, then **handler-case** returns the values resulting from it.

If *var* is unneeded, it may be omitted. For example, a clause such as:

```
    (type (var) (declare (ignore var)) form)
```

may be written more easily in the following way:

```
    (type () form)
```

A clause may have no forms after the argument specification. In this case, if the system transfers control to this clause, it returns `nil`.

The signalling process searches the clauses from top to bottom, as if the textually earlier clauses were dynamically bound later in time than the textually later clauses. This is analogous to **`typecase`**. If there is a type overlap, **`signal`** transfers control to the textually first case by way of invoking the handler. For purposes of invoking any one handler case, the dynamic context of bound handlers excludes all handlers established by a single **`handler-case`**.

As a special case, the *type* can be the symbol `:no-error` in the last clause. If the user specifies this, it designates a case that executes if *form* returns normally, and the arguments passed to the case are those returned by *form*.

Examples of **`handler-case`**:

```
    (handler-case (/ x y)
      (division-by-zero () nil))

    (handler-case (open *the-file* :direction :input)
      (file-error (condition)
        (format t "~&Fooey: ~A~%" condition)
        nil))

    (handler-case (some-user-function)
      (file-error (condition) condition)
      (division-by-zero () 0)
      ((or unbound-variable undefined-function) () 'unbound))

    (handler-case (intern x y)
      (error (condition) condition)
      (:no-error (symbol status)
                 (declare (ignore symbol))
                 status))
```

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

This macro executes *forms* in a context that handles conditions of type **`error`** by returning from this form two values: `nil` and the signaled condition object. If the system does not invoke this handler, either because no one signaled an **`error`** condition or because a tighter bound handler took care of the error, **`ignore-errors`** returns any values returned by the last form executed.

This is equivalent to the following:

```
    (handler-case (progn forms)
      (error (condition) (values nil condition)))
```

`handler-bind` ({ (*type handler*) }*) {*form*}* [*Macro*]

This macro executes its body in a dynamic context where the given handler bindings are in effect. *Type* may be any type specifier. *Handler* should evaluate to a function used to handle conditions of the associated type(s) during execution of the body. This function takes a required argument that is the signaled condition.

The signalling process searches the bindings from top to bottom, as if the textually earlier bindings were dynamically bound later in time than the textually later bindings. This is consistent with

handler-case which is analogous to **typecase**. If there is a type overlap, **signal** finds the earlier binding first. For purposes of invoking any one handler, the dynamic context of bound handlers excludes all handlers established by a single **handler-bind**.

If the body executes normally, this returns the values of the last *form*.

4.5. Defining and Creating Conditions

define-condition *name* (*parent-type*) [({*slot*}*) {*option*}*] [*Macro*]

This macro defines a new condition type called *name*, which is a subtype of the given *parent-type*. Except as otherwise noted, the expansion of this macro does not evaluate the arguments.

Objects of this condition type include slots available in objects of *parent-type* in addition to the indicated slots. A *slot* description has the following form:

{ *slot-name* | (*slot-name*) | (*slot-name* *default-value*) }

The *default-value* is a form evaluated by **make-condition** to produce a default value when the caller leaves the value unspecified. If the description of the slot does not provide a *default-value*, and the user of **make-condition** does not specify a value, then the system initializes the slot in an implementation-dependent way. It is an error to attempt to access a slot which has not been explicitly initialized and which has not been given a default value.

If the new type and some other type from which it inherits have a slot with the same name, **define-condition** only allocates one slot for the new type. Any specified default overrides any inherited default.

make-condition accepts keywords (in the keyword package) with the same name as any slot name and initializes the corresponding slot in conditions it creates.

Accessors are created according to the same rules used by **defstruct**, but it is an error to attempt to assign a condition's slots with **setf**. **Define-condition** interns accessor names into the package that is current when it executes.

A valid *option* is one of the following:

(:documentation *doc-string***)**
Doc-string is a string describing the purpose of the condition type or **nil**. If this option is unspecified, **define-condition** assumes **nil**. The documentation is retrievable with **(documentation name 'type)**, where **type** is the condition *name*.

(:conc-name *symbol-or-string***)**
As with **defstruct**, this sets up automatic prefixing of the names of slot accessors. The default is to use the name of the new type, *name*, followed by a hyphen.

(:report *exp***)** If *exp* is not a literal string, it must be a suitable argument to the **function** special form, and when **define-condition** expands, it evaluates the expression **(function** *exp***)** in the current lexical environment. The function takes two required arguments, a condition and a stream, and the system invokes the function whenever it prints the condition with ***print-escape*** bound to **nil**. See section 4.2.15. If *exp* is a literal string, it is a shorthand for the following:

```
(lambda (condition stream)
  (declare (ignore condition))
  (write-string exp stream))
```

This option inherits from *parent-type* if not specified.

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

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 (lambda (condition stream)
             (format stream "The machine ~A is not available."
                     (machine-error-machine-name condition)))))
```

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

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 this example leaves the **:report** option unspecified, it inherits the report method for **machine-unavailable** conditions.

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

This function constructs a condition object of type *type* using *slot-initializations*. This returns the condition object. *Slot-initializations* is given as alternating keyword/value pairs. The following example shows the creation of a condition type **peg/hole-mismatch** with slots named **peg-shape** and **hole-shape**:

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

4.6. Assertions

check-type *place typespec* **&optional** *string* [*Macro*]

This macro signals an error of type **type-error** if the contents of *place* are not of type *typespec*. If this signals a condition, handlers can use the functions **type-error-object** and **type-error-expected-type** to access the contents of *place* the desired type *typespec*, respectively. This form only returns if a handler, or the user from the debugger, invokes the **store-value** restart.

In this situation **store-value** takes an argument or prompts the user for it and stores the value in *place*, continuing within **check-type** which starts over possibly signalling an error again. Subforms of *place* may evaluate multiple times because of the implicit loop generated. **Check-type** returns **nil**.

Place must be a generalized variable reference acceptable to **setf**. *Typespec* must be a type specifier, and **check-type** does not evaluate it. *String* is a string literal describing type, and if it is unsupplied, **check-type** computes a description from *typespec*.

The error message will mention *place*, its contents, and the desired type.

Here are a couple examples:

```
* (setf aardvarks '(sam harry fred))
(SAM HARRY FRED)
* (check-type aardvarks (array * (3)))

The value of AARDVARKS is (SAM HARRY FRED),
 which is not of type (ARRAY * (3)).

Restarts:
  0: Supply a new value of AARDVARKS.
  1: Return to Top-Level.

Debug  (type H for help)
(LISP::CHECK-TYPE-ERROR AARDVARKS (SAM HARRY FRED)
                        (ARRAY * (3)) NIL)
0] restart 0
Type a form to be evaluated:
'#(sam fred harry)
NIL
* aardvarks
#(SAM FRED HARRY)
* (setf count 'foo)
FOO
* (check-type count (integer 0 *) "a positive integer")

The value of COUNT is FOO, which is not a positive integer.

Restarts:
  0: Supply a new value of COUNT.
  1: Return to Top-Level.

Debug  (type H for help)
(LISP::CHECK-TYPE-ERROR COUNT FOO
                        (INTEGER 0 *) "a positive integer")
0] restart 1
*
```

`assert` *test-form* `&optional` ({*place*}*) *datum* {*argument*}* [*Macro*]

This macro signals an error if the value of *test-form* is `nil`. Using the the `continue` restart allows the user to alter the values of some variables. If `continue` does execute, `assert` starts over, evaluating *test-form* and possibly signalling an error again. `Assert` returns `nil`.

Test-form is any form. Each *place* must be a generalized variable reference acceptable to `setf`. `Assert` only evaluates subforms of each *place* if the `continue` restart runs, and it may re-evaluate them each time the assertion fails.

This only evaluates *datum* and each *argument* if it signals a condition, and it will re-evaluate them each time the assertion fails. `Assert` uses these parameters in the following way:

- If *datum* is a condition, this uses it directly. In this case, it is an error to specify any *argument*.
- If *datum* is a condition type, then this uses the condition resulting from `apply`'ing `make-condition` to *datum* and *arguments*.
- If *datum* is a string, then this uses the condition resulting from the following:

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

- If *datum* is unsupplied, then this uses a condition of type **simple-error** constructed with *test-form* as data, for example:

```
(make-condition 'simple-error
                :format-string "The assertion ~S failed."
                :format-arguments '(test-form))
```

Here is an example of **assert**:

```
* (setf x (make-array '(3 5) :initial-element 3))
#2A((3 3 3 3 3) (3 3 3 3 3) (3 3 3 3 3))
* (setf y (make-array '(3 5) :initial-element 7))
#2A((7 7 7 7 7) (7 7 7 7 7) (7 7 7 7 7))
* (defun matrix-multiply (a b)
    (let ((*print-array* nil))
      (assert (and (= (array-rank a) (array-rank b) 2)
                   (= (array-dimension a 1) (array-dimension b 0)))
              (a b)
              "Cannot multiply ~S by ~S." a b)
      (really-matrix-multiply a b)))
MATRIX-MULTIPLY
* (matrix-multiply x y)

Error in function LISP::ASSERT-ERROR.
Cannot multiply #<Array, rank 2 {B8}> by #<Array, rank 2 {D4}>.

Restarts:
  0: Retry assertion with new values for A, B.
  1: Return to Top-Level.

Debug  (type H for help)
(LISP::ASSERT-ERROR (AND (= (# #) (# #) 2) (= (# # #) (# # #)))
                    (A B)
                    "Cannot multiply ~S by ~S."
                    #<Array, rank 2>...)
0] restart 0
The old value of A is #<Array, rank 2>.
Do you want to supply a new value? y
Type a form to be evaluated:
x
The old value of B is #<Array, rank 2>.
Do you want to supply a new value? y
Type a form to be evaluated:
(make-array '(5 3) :initial-element 6)
#2A((54 54 54 54 54)
    (54 54 54 54 54)
    (54 54 54 54 54)
    (54 54 54 54 54)
    (54 54 54 54 54))
*
```

4.7. Case Forms

This section describes case forms similar to **case** and **typecase** that signal an error if no branch fires.

etypecase *keyform* {(*type* {*form*}*)}* *[Macro]*

This control construct is similar to **typecase**, but no explicit **otherwise** or **t** clause is permitted. If no clause fires, this signals an error of type **type-error** with a message constructed from the clauses. This error cannot be continued. To supply your own error message, use **typecase** with an **otherwise** clause containing a call to **error**. The name of this function stands for "exhaustive type case" or "error-checking type case."

Here is an example:

```
* (setq x 1/3)
1/3
* (etypecase x
    (integer (* x 4))
    (symbol  (symbol-value x)))

1/3 fell through ETYPECASE expression.
Wanted one of (SYMBOL INTEGER).

Restarts:
  0: Return to Top-Level.

Debug  (type H for help)
(LISP::%EVAL (ERROR 'CONDITIONS::CASE-FAILURE
                    :NAME 'ETYPECASE :DATUM ...))
0]
```

ctypecase *keyplace* {(*type* {*form*}*)}* *[Function]*

This control construct is similar to **typecase**, but no explicit **otherwise** or **t** clause is permitted. The *keyplace* must be a generalized variable reference acceptable to **setf**. If no clause fires, **ctypecase** signals an error of type **type-error** with a message constructed from the clauses. The user may continue this error using the **store-value** restart.

In this situation **store-value** takes an argument or prompts the user for it and stores the value in *keyplace*, continuing within **ctypecase** which starts over possibly signalling an error again. Subforms of *keyplace* may evaluate multiple times because of the implicit loop generated.

This returns any values returned by the last *form* in the selected case. The name of this function is mnemonic for "continuable (exhaustive) type case."

Here is an example:

```
* (setq x 1/3)
1/3
* (ctypecase x
             (integer (* x 4))
             (symbol  (symbol-value x)))

1/3 fell through CTYPECASE expression.
Wanted one of (SYMBOL INTEGER).

Restarts:
  0: Supply a new value for X.
  1: Return to Top-Level.

Debug  (type H for help)
(LISP::CASE-BODY-ERROR CTYPECASE X 1/3 (OR SYMBOL INTEGER)...)
0] restart 0
Type a form to be evaluated:
3.7

3.7 fell through CTYPECASE expression.
Wanted one of (SYMBOL INTEGER).

Restarts:
  0: Supply a new value for X.
  1: Return to Top-Level.

Debug  (type H for help)
(LISP::CASE-BODY-ERROR CTYPECASE X 3.699997 (OR SYMBOL INTEGER)...)
0] restart 0
Type a form to be evaluated:
12
48
*
```

ecase *keyform* {*case*}* [*Macro*]

This control construct is similar to **case**, but no explicit **otherwise** or **t** clause is permitted. Each *case* is of the following form:

({ (*key1* *key2* ...) | *key* } *form1* *form2* ...)

If no case fires, this signals an error of type **type-error** with a message constructed from the cases. This error cannot be continued. To supply your own error message, use **case** with an **otherwise** clause containing a call to **error**. The name of this function stands for "exhaustive case" or "error-checking case."

Here is an example:

```
* (setq x 1/3)
1/3
* (ecase x
    (alpha (foo))
    (omega (bar))
    ((zeta phi) (baz)))
1/3 fell through ECASE expression.
Wanted one of (ZETA PHI OMEGA ALPHA).

Restarts:
  0: Return to Top-Level.

Debug  (type H for help)
(LISP::%EVAL (ERROR 'CONDITIONS::CASE-FAILURE
                    :NAME 'ECASE :DATUM ...))
0]
```

ccase *keyplace* {*case*}* [*Macro*]

This control construct is similar to **case**, but no explicit **otherwise** or **t** clause is permitted. The *keyplace* must be a generalized variable reference acceptable to **setf**. Each *case* is of the following form:

({ (*key1 key2* ...) | *key* } *form1 form2* ...)

If no clause fires, **ccase** signals an error of type **type-error** with a message constructed from the clauses. The user may continue this error using the **store-value** restart.

In this situation **store-value** takes an argument or prompts the user for it and stores the value in *keyplace*, continuing within **ccase** which starts over possibly signalling an error again. Subforms of *keyplace* may evaluate multiple times because of the implicit loop generated.

This returns any values returned by the last *form* in the selected case. The name of this function is mnemonic for "continuable (exhaustive) case."

4.8. Establishing Restarts

with-simple-restart (*name format-string* **&rest** *format-arguments*) {*form*}* [*Macro*]

This macro is shorthand for a common use of **restart-case**. *Name* is the name of the restart, and if this one executes, control returns to **with-simple-restart** returning the values **nil** and **t**. If each *form* executes normally, then the values of the last one are returned.

Name may be **nil**, in which case, this establishes an anonymous restart.

By way of example, you could define **with-simple-restart** in the following way:

```
(defmacro with-simple-restart ((restart-name format-string
                                  &rest format-arguments)
                                 &body forms)
  `(restart-case (progn ,@forms)
     (,restart-name ()
       :report (lambda (stream)
                 (format stream ,format-string ,@format-arguments))
       (values nil t))))
```

Here is an example of its use:

```
* (defun read-eval-print-loop (level)
    (with-simple-restart (abort "Exit command level ~D." level)
      (loop
        (with-simple-restart (abort "Return to command level ~D."
                                    level)
          (let ((form (prog2 (fresh-line) (read) (fresh-line))))
            (prin1 (eval form)))))))
READ-EVAL-PRINT-LOOP
* (read-eval-print-loop 1)
(+ 'a 3)

Error in function +.
Wrong type argument, A, should have been of type NUMBER.

Restarts:
  0: Return to command level 1.
  1: Exit command level 1.
  2: Return to Top-Level.

Debug  (type H for help)
(CONDITIONS::MAKE-ERROR-TABLE + 0 A NIL)
0] restart 0
(+ 5 nil)

Error in function +.
Wrong type argument, NIL, should have been of type NUMBER.

Restarts:
  0: Return to command level 1.
  1: Exit command level 1.
  2: Return to Top-Level.

Debug  (type H for help)
(CONDITIONS::MAKE-ERROR-TABLE + 0 NIL NIL)
0] restart 1
NIL
T
*
```

restart-case *expression* { (*case-name arglist* {*keyword value*}* {*form*}*) }* [*Macro*]

This macro evaluates *expression* in a dynamic context where the clauses have special meanings as points to which handlers and users may transfer program control. If *expression* executes normally, **restart-case** returns any values returned by it. If anyone invokes one of the restarts, the system transfers control to that branch executing each *form* and returning any values returned by the last such *form*.

If there are no forms in a selected clause, **restart-case** returns **nil**.

Case-name may be **nil** or a symbol naming the restart. A *case-name* may be repeated, in which case **find-restart** will find the first such clause that appears textually. The other clauses are accessible using **compute-restarts**.

Each *arglist* is a normal lambda list of locals to be bound during the execution of its corresponding forms. These arguments convey any necessary data from a call to **invoke-restart** to the clause.

Valid *keyword*/*value* pairs are as follows:

:interactive exp

By default, **invoke-restart-interactively** passes no arguments to a restart, and all the arguments must be optional to accomodate interactive restarting, what typically occurs in the debugger with user intervention. The arguments may be required if the restart specifies the **:interactive** *keyword*, and **exp** must be a suitable argument to the **function** special form. **Restart-case** evaluates the expression (**function exp**) in the current lexical environment. It should return a function of no arguments that returns a list of values to which **invoke-restart-interactively** will apply the restart. This function runs in the dynamic environment available prior to any restart attempt. The interactive function may use the ***query-io*** stream.

:report exp

If **exp** is not a literal string, it must be a suitable argument to the **function** special form, and **restart-case** evaluates the expression (**function exp**) in the current lexical environment. The function takes one required argument, a stream, and the system invokes the function whenever it prints the restart with ***print-escape*** bound to **nil**. If **exp** is a literal string, it is a shorthand for the following:

```
(lambda (stream)
  (write-string exp stream))
```

If the system reports a named restart, and it has no report method, the system uses the restart name in generating default report text. It is an error to define an unnamed restart without any report information since these are generally only useful interactively, possibly as an option for the user in the debugger.

Here are some examples:

```
(loop
  (restart-case (return (apply function some-args))
    (new-function (new-function)
                  :report "Use a different function."
                  :interactive
                  (lambda
                    () (list (prompt-for 'function "Function: ")))
      (setq function new-function))))

(loop
  (restart-case (return (apply function some-args))
    (nil (new-function)
         :report "use a different function."
         :interactive
         (lambda () (list (prompt-for 'function "function: ")))
      (setq function new-function))))

(restart-case (a-command-loop)
  (return-from-command-level
      ()
      :report
      (lambda (stream)
        (format stream "Return from command level ~D." level))
    nil))

(loop
  (restart-case (another-random-computation)
    (continue ()
      nil)))
```

prompt-for is immaterial to this example and does what you think. The first and second examples are equivalent from the point of view of someone using the interactive debugger, but differ in one important aspect for non-interactive handling; a handler can make use of named restarts other than **nil** as in the following piece of code:

```
    (if (find-restart 'new-function)
        (invoke-restart 'new-function the-replacement))
```

This works for the first one, but the second one is only callable interactively, such as from the debugger.

Here is a more complete example:

```
    (let ((my-food 'milk)
          (my-color 'greenish-blue))
      (do ()
          ((not (bad-food-color-p my-food my-color)))
        (restart-case (error 'bad-food-color
                             :food my-food :color my-color)
          (use-food  (new-food)
                     :report "Use another food."
                     (setq my-food new-food))
          (use-color (new-color)
                     :report "Use another color."
                     (setq my-color new-color))))
      ;; Can't get here until my-food and my-color are compatible.
      (list my-food my-color))
```

Handlers written for **bad-food-color** errors can use the **find-restart/invoke-restart** idiom to supply a different food for the given color or a different color for the given food. See section 4.2.7 for a discussion of encapsulating this idiom for programmer and user convenience.

restart-bind ({*binding*}*) {*form*}* [*Function*]

This macro executes each *form* in a dynamic context where the given restart bindings are in effect. Each binding is of the following form:

(*name function* {*keyword value*}*)

Name may be **nil** to indicate an anonymous restart, or some other symbol to indicate a named restart. *Function* should evaluate to a function that performs the restart. If invoked, this function either transfers control non-locally or simply returns, and it takes whatever arguments the programmer desires. **Invoke-restart** and **invoke-restart-interactively** are the only ways to call it, either from a piece of code or as the result of a debugger command. In the case of interactive invocation, where the arguments are not supplied, the second function named calls the **:interactive-function** option (see below).

The valid *keyword/value* pairs are:

:interactive-function *form*
: The *form* evaluates in the current lexical environment and should return a function of no arguments that returns a list of arguments to which **invoke-restart-interactively** applies the restart function. The function may prompt using ***query-io***.

:report-function *form*
: The *form* evaluates in the current lexical environment and should return a function that takes a stream as an argument and prints on it a summary of the action that this restart will take. The system calls this function whenever the restart is printed with ***print-escape*** bound to **nil**.

This is considered a significantly lower-level primitive than **restart-case**, and its intended purpose is that of building higher-level abstractions such as **restart-case**. It still has uses for inclusion in general coding, but typically it appears in macros that define other constructs.

4.9. Finding and Manipulating Restarts

`compute-restarts` *[Function]*

This function returns a list of the restarts currently active in the dynamic state of a program, see **`restart-bind`** and **`restart-case`**. Each restart represents a function that performs some recovery action, typically a dynamic transfer of control. Restart objects are implementation-dependent, but they always have dynamic extent relative to the scope of the binding form.

The result of **`compute-restarts`** is ordered from more recently established restarts to those first established in time. All elements of the list are valid, including anonymous restarts, even though some may have the same name as others and would not be found by **`find-restart`** because of this.

Portable programs do not rely on whether multiple calls to **`compute-restarts`** in the same dynamic environment share elements or are disjoint (not **`eq`**), and it is an error to modify the resulting list.

`restart-name` *restart* *[Function]*

This function returns the name of the given **`restart`** object. If it is unnamed, this returns **`nil`**.

`find-restart` *identifier* *[Function]*

This function searches for a particular restart in the current dynamic environment.

If *identifier* is a symbol, this returns the most recently established restart with that name. If none is found, this returns **`nil`**.

If *identifier* is a **`restart`** object, this returns the object if it is currently active. If it is inactive, this returns **`nil`**.

Although anonymous restarts have **`nil`** as a name, it is an error to supply the symbol **`nil`** for *identifier*. If your application seems to require this, consider rewriting it to use **`compute-restarts`**.

`invoke-restart` *restart* **`&rest`** *arguments* *[Function]*

This function calls the function associated with *restart* on *arguments*. *Restart* must be a **`restart`** object or the non-**`nil`** name of a currently valid restart. If the argument is invalid, this signals a **`control-error`** error. Note, restart functions (see section 4.2.7), such as **`abort`** and **`continue`**, call this function, not vice versa.

`invoke-restart-interactively` *restart* *[Function]*

This function invokes the function associated with *restart*. If *restart* has an associated interactive method (see **`restart-bind`** (page 46) and **`restart-case`** (page 44)), this function calls the method to provide arguments for *restart*'s function. *Restart* must be a **`restart`** object or the non-**`nil`** name of a restart that is valid in the current dynamic context. If it is invalid, this function signals an error of type **`control-error`**.

If no interactive method is associated with *restart*, then it is an error for the *restart*'s function to require arguments.

4.10. Restart Functions

abort *[Function]*

This function transfers control to the restart named **abort**, and if none exists, it signals an error of type **control-error**. This is generally used to return to previous command levels.

continue *[Function]*

This function transfers control to the restart named **continue**, and if none exists, it returns **nil**. This is generally used with simple and *obvious* restarts, such as in **break** and **cerror**, whether in user or system code.

muffle-warning *[Function]*

This function transfers control to the restart named **muffle-warning**, and if none exists, it signals an error of type **control-error**. **Warn** signals **warning** conditions in an environment where this restart causes **warn** to immediately return.

store-value *value* *[Function]*

This function transfers control, passing *value*, to the restart named **store-value**, and if none exists, it returns **nil**. Code that signals errors of type **cell-error** and **type-error** may establish this restart for handlers that can supply replacement data to be stored permanently to correct the situation.

use-value *value* *[Function]*

This function transfers control, passing *value*, to the restart named **use-value**, and if none exists, it returns **nil**. Code that signals **cell-error** errors may establish this restart for handlers that can supply a replacement value to be used once only to correct the situation.

4.11. Debugging Utilities

break &optional *format-string* **&rest** *format-arguments* *[Function]*

This function prints the message described by *format-string* and *format-arguments* and then enters the debugger. While in the debugger, there is a **continue** restart that causes **break** to return **nil** immediately. If *format-string* is unsupplied, this generates a default message.

By way of example, **break** could be defined as follows:

```
(defun break (&optional (format-string "Break")
                        &rest format-arguments)
  (with-simple-restart (continue "Return from BREAK.")
    (invoke-debugger
      (make-condition 'simple-condition
                      :format-string format-string
                      :format-arguments format-arguments)))
    nil)
```

invoke-debugger *condition* *[Function]*
debugger-hook *[Variable]*

This function invokes an interactive mechanism for handling *condition*, which must be a **condition** object. This never directly returns; some non-local transfer of control must occur, such as the use of a restart, aborting to top level, etc.

When the variable ***debugger-hook*** is non-**nil**, it is a function **invoke-debugger** calls instead of executing any standard debugger interface. The function takes *condition* and the value of ***debugger-hook*** as arguments, and if it returns, **invoke-debugger** enters the standard debugger anyway. While executing ***debugger-hook***, this variable is **nil**, so if this interface evaluates code on the user's behalf, it may want to rebind `*debugger-hook*` to the second value passed in to handle recursive errors with the same interface.

4.12. System Defined Types

`Restart` is the data type used to represent a restart.

A sketch of the `condition` type hierarchy looks like this:

```
                              CONDITION
                                  |
        +-----------------+--------------------+---  . . .
        |                 |                    |
        |                 |                    |
 SIMPLE-CONDITION   SERIOUS-CONDITION      WARNING
                          |                    |
                          |                    |
                          |             SIMPLE-WARNING
                          |
                          |
                    +-----+-----------------------+---  . . .
                    |                             |
                    |                             |
                  ERROR                  STORAGE-CONDITION
                    |
                    |
     +--------------+---+--------------+---  . . .
     |              |                  |
     |              |                  |
SIMPLE-ERROR   ARITHMETIC-ERROR   CONTROL-ERROR
                    |                  |
                   ...                ...
```

Typically programs do not directly instantiate conditions of the non-terminal types in the above tree (for example `condition`, `warning`, `storage-condition`, `error`, `arithmetic-error`, etc.); the system provides these primarily for type inclusion purposes.

The design of the condition system permits implementations to support non-portable synonyms for these types, as well as to introduce other types above, below, or between the types shown in this tree as long as the indicated subtype relationships are not violated.

The types `simple-condition`, `serious-condition`, and `warning` are pairwise disjoint. The type `error` is disjoint from types `simple-condition` and `warning`.

The following describes all the predefined condition types:

`condition`	All types of conditions, whether `error` or non-`error`, must inherit from this type.
`warning`	All types of warnings inherit from this type. This is a subtype of `condition`.
`serious-condition`	All serious conditions (conditions serious enough to require interactive intervention if not handled) inherit from this type. This is a subtype of `condition`.
`error`	All types of `error` conditions inherit from this condition. This is a subtype of `serious-condition`.
`simple-condition`	

Conditions signalled by **signal** when given a **format** string as a first argument are of this type. This is a subtype of **condition**. The system supports the initialization keywords **:format-string** and **:format-arguments** for the slots, which can be accessed using **simple-condition-format-string** and **simple-condition-format-arguments**. If **:format-arguments** is unsupplied with **make-condition**, the **format-arguments** slot defaults to **nil**.

simple-warning
Conditions signalled by **warn** when given a **format** string as a first argument are of this type. This is a subtype of **warning**. The system supports the initialization keywords **:format-string** and **:format-arguments** for the slots, which can be accessed using **simple-condition-format-string** and **simple-condition-format-arguments**. If **:format-arguments** is unsupplied with **make-condition**, the **format-arguments** slot defaults to **nil**. In implementations supporting multiple inheritance, this type will also be a subtype of **simple-condition**.

simple-error Conditions signalled by **error** and **cerror** when given a **format** string as a first argument are of this type. This is a subtype of **error**. The system supports the initialization keywords **:format-string** and **:format-arguments** for the slots, which can be accessed using **simple-condition-format-string** and **simple-condition-format-arguments**. If **:format-arguments** is unsupplied with **make-condition**, the **format-arguments** slot defaults to **nil**. In implementations supporting multiple inheritance, this type will also be a subtype of **simple-condition**.

storage-condition
Conditions related to storage overflow inherit from this type. This is a subtype of **serious-condition**.

type-error Errors in the transfer of data in a program inherit from this type. This is a subtype of **error**. For example, conditions signalled by **check-type** inherit from this type. The system supports the initialization keywords **:datum** and **:expected-type** for the slots, which can be accessed using **type-error-datum** and **type-error-expected-type**.

simple-type-error
Conditions signalled by facilities similar to **check-type** may use this type. The system supports the initialization keywords **:format-string** and **:format-arguments** for the slots, which can be accessed using **simple-condition-format-string** and **simple-condition-format-arguments**. If **:format-arguments** is unsupplied with **make-condition**, the **format-arguments** slot defaults to **nil**. In implementations supporting multiple inheritance, this type will also be a subtype of **simple-condition**.

program-error Errors related to incorrect program syntax statically detectable inherit from this type, regardless of whether they are statically detected. This is a subtype of **error**. This is **not** a subtype of **control-error**. The errors resulting from naming a go tag or **return-from** tag which is not lexically apparent are program errors.

control-error Errors in the dynamic transfer of control in a program inherit from this type. This is a subtype of **error**. This is **not** a subtype of **program-error**. The errors resulting from giving **throw** a tag which is not active or from giving go or **return-from** a tag which is no longer dynamically available are control errors.

package-error Errors occurring during operations on packages inherit from this type. This is a subtype of **error**. The system supports the initialization keyword **:package** for the slot, which can be accessed using **package-error-package**.

stream-error Errors occurring during input from, output to, or closing a stream inherit from this type. This is a subtype of **error**. The system supports the initialization keyword **:stream** for the slot, which can be accessed using **stream-error-stream**.

end-of-file The error resulting when reading from a stream with no more input inherits from this type. This is a subtype of **stream-error**.

file-error Errors occurring during an attempt to open a file, or during some low-level transaction with a file system, inherit from this type. This is a subtype of **error**. The system supports the initialization keyword **:pathname** for the slot, which can be accessed using

file-error-pathname.

cell-error Errors occurring while accessing a location inherit from this type. This is a subtype of **error**. The system supports the initialization keyword **:name** for the slot, which can be accessed using **cell-error-name**.

unbound-variable
The error resulting from trying to access the value of an unbound variable inherits from this type. This is a subtype of **cell-error**.

undefined-function
The error resulting from trying to access the value of an undefined function inherit from this type. This is a subtype of **cell-error**.

arithmetic-error
Errors occurring while doing arithmetic type operations inherit from this type. This is a subtype of **error**. The system supports the initialization keywords **:operation** and **:operands** for the slots, which can be accessed using **arithmetic-error-operation** and **arithmetic-error-operands**.

division-by-zero
Errors occurring because of division by zero inherit from this type. This is a subtype of **arithmetic-error**.

floating-point-overflow
Errors occurring because of floating point overflow inherit from this type. This is a subtype of **arithmetic-error**.

floating-point-underflow
Errors occurring because of floating point underflow inherit from this type. This is a subtype of **arithmetic-error**.

Chapter 5

Debugging Tools

By Jim Large, Steve Handerson, and Bill Chiles

5.1. 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, **`trace`** returns 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*` are untraced.

extensions:*traced-function-list* [*Variable*]

A list of function names maintained and used by `trace`, `untrace`, and `untrace-all`. This list should contain the names of all functions 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 40.

5.1.1. Encapsulation Functions

The encapsulation[2] functions provide a clean mechanism for intercepting the arguments and results of a function. **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 are 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**.

5.2. The Single Stepper

step *form* [*Macro*]

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 while printing the current form. ***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.

5.3. 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.

5.3.1. 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.

5.3.2. 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.

5.3.3. 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 Prints the condition given to `debug` and the 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.

5.3.4. 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.

5.3.5. 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 6

The Compiler

6.1. Calling the Compiler

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

`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: the Language*. 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 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.

6.2. 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)).
```

6.3. Compiler Switches

Several compiler switches are available which are not documented in the *Common Lisp: the Language*. 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`.

6.4. 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 7

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.

7.1. 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.

7.2. 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.

7.3. 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 7.2).
- 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 your heart of hearts that iterative Lisp is beautiful then you can join the ranks of the Lisp efficiency fiends.

7.4. 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 *Common Lisp: the Language*. 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.

7.4.1. 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 7.2).
- 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.

7.4.2. 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.

7.4.3. 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.

7.4.4. 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.

7.5. 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.

7.6. 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 is 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.

7.7. 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 have 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.

7.8. 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.

7.9. 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 them 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 8

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.

8.1. 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.

8.2. 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 83.) 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 want all addresses specified as integers). The function **`int-sap`** is used to do the opposite conversion.

8.3. 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*.

8.4. 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>
*
```

8.5. 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 CLX interface to 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" "XLIB" "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.
XLIB	Holds the CLX interface to the X window manager version 11. See the CLX documentation for details.
TS, **EVAL**	Holds Matchmaker interfaces used to control Lisp client processes from Hemlock.

8.6. 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.

8.7. 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 in ***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.

8.8. 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.

8.9. 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 77). 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** (page 13)) may be used to execute forms with the interrupt system effectively turned off and should be used in such situations.

Chapter 9

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.

9.1. 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.

9.2. The SERVER Function

The **server** function is the standard way for an application to wait for something to happen. For example, the Lisp calls **server** when it wants input from X or an ASCII terminal. The idea behind **server** is that it 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. Note that if an application is waiting for input of any kind, then other applications will get events.

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 69) 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 goes into an input wait.

system:serve-all &optional *timeout* [*Function*]

The function **serve-all** is similar to server, except that it serves all the pending events rather than just one. It returns T if at least one event is serviced and NIL otherwise.

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.

9.3. 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.

9.4. 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.

9.5. A SERVER Example

This section 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" "XLIB"))

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

(defun key-pressed (hunk event-key event-window root child
                         same-screen-p x y root-x root-y
                         modifiers time key-code send-event-p)
  "Key-pressed is called when a key press event is generated by X."
  (declare (ignore hunk event-key root child same-screen-p x y
                   root-x root-y modifiers time send-event-p))
  (format t "Key-pressed (Window = ~D) = ~D.~%"
          (xlib:window-id event-window)
          key-code))

(ext:serve-key-press  *xwindows* #'key-pressed)

(defun server-example ()
  "An example of using the server function."
  ;; Open the display and create an X window.
  (let* ((display (ext:open-clx-display))
         (screen (display-default-screen display))
         (black (screen-black-pixel screen))
         (white (screen-white-pixel screen))
         (window (create-window :parent (screen-root screen)
                                :background black
                                :border white
                                :border-width 5
                                :x 0
                                :y 0
                                :width 200
                                :height 200
                                :event-mask '(:key-press))))
    ;; Wrap code in unwind-protect, so we clean up after ourselves.
    (unwind-protect
        (progn
          ;; map the window to the screen.
          (map-window window)
          ;; Add the window to the *xwindows* object set.
          (system:add-xwindow-object window window *xwindows*)
          ;; Make sure the window gets displayed.
          (display-force-output display)
          ;; Enable event handling on the display.
          (ext:enable-clx-event-handling display
                                         #'ext:object-set-event-handler)
          ;; Call server for 100,000 events.
          (dotimes (i 100000) (system:server)))
      ;; Disable event handling on this display.
      (ext:disable-clx-event-handling display)
      ;; Remove the window from the object set before destroying it.
      (system:remove-xwindow-object window)
      ;; Destroy the window.
      (destroy-window window)
      ;; Make sure X destroys window NOW.
      (display-force-output display)
      ;; Close the display.
      (xlib:close-display display))))
```

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 10

The Alien Facility

By Rob Maclachlan

10.1. 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.

10.2. 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.

10.3. 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.

10.4. 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 85).

`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))
```

10.5. Alien Variables

An **`Alien`** variable is a symbol that has 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 is non-nil, 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 84). *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.

10.6. 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*.

10.7. 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 86). 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.

10.8. 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 11

Foreign Function Call Interface

By David B. McDonald

11.1. 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.

11.2. 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.

11.3. 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.

11.3.1. 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))
```

11.3.2. 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 an 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)
```

11.3.3. 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) 'alien) cs)
```

11.3.4. 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)
```

11.4. 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)
```

11.5. 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 on to C. *Options* is currently ignored. *Amode* should be one of the following:

:in	This specifies that the argument is passed by value. This 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))
```

11.6. 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.
```

11.7. 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 12

User-Defined Assembler Language Routines

By David B. McDonald

12.1. 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.

12.2. 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.

12.3. 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.

12.4. 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 a listing file (with default extension s) should generated that is 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 assumed to be 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.

12.5. 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.

12.5.1. 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 is.

12.5.2. 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.

12.5.3. 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.

12.5.4. 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.

12.5.5. 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.

12.5.6. 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** $\mathbf{R_1}$ **I4**)
The value I4 is subtracted from R_1. Condition code bits lt, eq, gt, c0, and ov are modified.

Divide step: (**d** $\mathbf{R_1}$ $\mathbf{R_2}$)
If you really want to do a divide step see *IBM RT PC Hardware Technical Reference Manual*. See section 12.7 for a miscop routine that does division.

Multiply step: (**m** $\mathbf{R_1}$ $\mathbf{R_2}$)
Again, you don't really want to use this instruction. See 12.7 for a miscop that does multiplication.

12.5.7. 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** $\mathbf{R_1}$ **I4**)
The bit specified by I4 in the lower half of R_1 is set to 0.

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

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

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

And: (**n** $\mathbf{R_1}$ $\mathbf{R_2}$) The logical and of R_1 and R_2 replaces the contents of R_1:

And immediate lower half extended zeroes: (**nilz** $\mathbf{R_1}$ $\mathbf{R_2}$ **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** $\mathbf{R_1}$ $\mathbf{R_2}$ **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** $\mathbf{R_1}$ $\mathbf{R_2}$ **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** $\mathbf{R_1}$ $\mathbf{R_2}$ **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** $\mathbf{R_1}$ $\mathbf{R_2}$) The logical or of R_1 and R_2 replaces the contents of R_1.

Or immediate lower: (**oil** $\mathbf{R_1}$ $\mathbf{R_2}$ **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** $\mathbf{R_1}$ $\mathbf{R_2}$ **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** $\mathbf{R_1}$ $\mathbf{R_2}$)
The logical exclusive or of R_1 and R_2 replaces the contents of R_1.

Exclusive or immediate lower half: (**xil** $\mathbf{R_1}$ $\mathbf{R_2}$ **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** $\mathbf{R_1}$ $\mathbf{R_2}$ **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** $\mathbf{R_1}$ $\mathbf{R_2}$)
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.

12.5.8. 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.

12.5.9. 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.

12.5.10. 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.

12.6. Useful Macros

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

12.6.1. 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.

12.6.2. 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.

12.6.3. 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.

12.6.4. 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**.

12.6.5. 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.

12.6.6. 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.

12.7. 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.

fpa-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.

fpa-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.

fpa-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.

fpa-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.

12.8. 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.

12.9. 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.

12.10. 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