CMUCL User's Manual

Robert A. MacLachlan, *Editor*

April 12, 2001

Abstract

CMUCL is a free, high-performance implementation of the Common Lisp programming language, which runs on most major Unix platforms. It mainly conforms to the ANSI Common Lisp Standard. CMUCL features a sophisticated native-code compiler, a foreign function interface, a graphical source-level debugger, an interface to the X11 Window System, and an Emacs-like editor.

Keywords: lisp, Common Lisp, manual, compiler, programming language implementation, programming environment

Contents

Chapter 1

Introduction

CMUCL is a free, high-performance implementation of the Common Lisp programming language which runs on most major Unix platforms. It mainly conforms to the ANSI Common Lisp standard. Here is a summary of its main features:

- a *sophisticated native-code compiler* which is capable of powerful type inferences, and generates code competitive in speed with C compilers.
- generational garbage collection and multiprocessing capability on the x86 ports.
- a foreign function interface which allows interfacing with C code and system libraries, including shared libraries on most platforms, and direct access to Unix system calls.
- support for interprocess communication and remote procedure calls.
- an implementation of CLOS, the Common Lisp Object System, which includes multimethods and a metaobject protocol.
- a graphical source-level debugger using a Motif interface, and a code profiler.
- an interface to the X11 Window System (CLX), and a sophisticated graphical widget library (Garnet).
- programmer-extensible input and output streams.
- an Emacs-like editor implemented in Common Lisp.
- public domain: free, with full source code and no strings attached (and no warranty). Like the GNU/Linux and *BSD operating systems, CMUCL is maintained and improved by a team of volunteers collaborating over the Internet.

This user's manual contains only implementation-specific information about CMUCL. Users will also need a separate manual describing the Common Lisp standard, for example the Hyperspec.

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

1.1 Distribution and Support

CMUCL is developed and maintained by a group of volunteers who collaborate over the internet. Sources and binary releases for the various supported platforms can be obtained from www.cons.org/cmucl. These pages describe how to download by FTP or CVS.

A number of mailing lists are available for users and developers; please see the web site for more information.

1.2 Command Line Options

The command line syntax and environment is described in the `lisp(1)` man page in the man/man1 directory of the distribution. See also `cmucl(1)`. Currently CMUCL accepts the following switches:

-batch specifies batch mode, where all input is directed from standard-input. An error code of 0 is returned upon encountering an EOF and 1 otherwise.

-quiet enters quiet mode. This implies setting the variables *load-verbose*, *compile-verbose*, *compile-print*, *compile-progress*, *require-verbose* and *gc-verbose* to NIL, and disables the printing of the startup banner.

-core requires an argument that should be the name of a core file. Rather than using the default core file ('`lib/lisp.core`'), the specified core file is loaded.

-edit specifies to enter Hemlock. 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.`*object-type*', 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.`*object-type*' 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.

-slave specifies that Lisp should start up as a ıslave Lisp and try to connect to an editor Lisp. The name of the editor to connect to must be specified—to find the editor's name, use the Hemlock "Accept Slave Connections" command. The name for the editor Lisp is of the form:

machine-name:*socket*

where *machine-name* is the internet host name for the machine and *socket* is the decimal number of the socket to connect to.

For more details on the use of the -edit and -slave 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 use either of the following two commands:

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

1.3 Credits

CMUCL was developed at the Computer Science Department of Carnegie Mellon University. The work was a small autonomous part within the Mach microkernel-based operating system project, and started more as a tool development effort than a research project. The project started out as Spice Lisp, which provided a modern Lisp implementation for use in the CMU community. CMUCL has been under continual development since the early 1980's (concurrent with the Common Lisp standardization effort). Most of the CMU Common Lisp implementors went on to work on the Gwydion environment for Dylan. The CMU team was lead by Scott E. Fahlman, the Python compiler was written by Robert MacLachlan.

CMUCL's CLOS implementation is derived from the PCL reference implementation written an Xerox PARC. Its implementation of the LOOP macro was derived from code from Symbolics, which was derived from code written at MIT. The CLX code is copyrighted by Texas Instruments Incorporated.

CMUCL was funded by DARPA under CMU's "Research on Parallel Computing" contract. Rather than doing pure research on programming languages and environments, the emphasis was on developing practical programming tools. Sometimes this required new technology, but much of the work was in creating a Common Lisp environment that incorporates state-of-the-art features from existing systems (both Lisp and non-Lisp). Archives of the project are available online.

The project funding stopped in 1994, so support at Carnegie Mellon University has been discontinued. The project continues as a group of users and developers collaborating over the Internet. The currently active maintainers are:

- Marco Antoniotti
- Martin Cracauer
- Douglas Thomas Crosher
- Rob MacLachlan
- Raymond Toy
- Peter Van Eynde
- Paul Werkowski

In particular Paul Werkowski and Douglas Crosher completed the port for the x86 architecture for FreeBSD. Peter VanEnyde took the FreeBSD port and created a Linux version. Other people who have contributed to the development of CMUCL since 1981 are

- David Axmark
- Miles Bader
- Rick Busdiecker
- Bill Chiles
- Casper Dik
- Ted Dunning
- Scott Fahlman
- Mike Garland
- Paul Gleichauf
- Sean Hallgren
- Richard Harris
- Joerg-Cyril Hoehl

- Chris Hoover
- John Kolojejchick
- Todd Kaufmann
- Simon Leinen
- Sandra Loosemore
- William Lott
- Dave McDonald
- Tim Moore
- Skef Wholey

This manual is based on CMU Technical Report CMU-CS-92-161, edited by Robert A. MacLachlan, dated July 1992. Other contributors include Raymond Toy, Paul Werkowski and Eric Marsden.

Chapter 2

Design Choices and Extensions

Several design choices in Common Lisp are left to the individual implementation, and some essential parts of the programming environment are left undefined. This chapter discusses the most important design choices and extensions.

2.1 Data Types

2.1.1 Symbols

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.

2.1.2 Integers

The fixnum type is equivalent to (signed-byte 30). Integers outside this range are represented as a bignum or a word integer (see section 5.11.6, page 87.) Almost all integers that appear in programs can be represented as a fixnum, so integer number consing is rare.

2.1.3 Floats

CMUCL supports two floating point formats: single-float and double-float. These are implemented with IEEE single and double float arithmetic, respectively. short-float is a synonym for single-float, and long-float is a synonym for double-float. The initial value of *read-default-float-format* is single-float.

Both single-float and double-float are represented with a pointer descriptor, so float operations can cause number consing. Number consing is greatly reduced if programs are written to allow the use of non-descriptor representations (see section 5.11, page 84.)

2.1.3.1 IEEE Special Values

CMUCL supports the IEEE infinity and NaN special values. These non-numeric values will only be generated when trapping is disabled for some floating point exception (see section 2.1.3.4, page 6), so users of the default configuration need not concern themselves with special values.

extensions:short-float-positive-infinity [*Constant*]
extensions:short-float-negative-infinity [*Constant*]
extensions:single-float-positive-infinity [*Constant*]
extensions:single-float-negative-infinity [*Constant*]
extensions:double-float-positive-infinity [*Constant*]
extensions:double-float-negative-infinity [*Constant*]
extensions:long-float-positive-infinity [*Constant*]

extensions:long-float-negative-infinity [*Constant*]

The values of these constants are the IEEE positive and negative infinity objects for each float format.

extensions:float-infinity-p *x* [*Function*]

This function returns true if *x* is an IEEE float infinity (of either sign.) *x* must be a float.

extensions:float-nan-p *x* [*Function*]

extensions:float-trapping-nan-p *x* [*Function*]

float-nan-p returns true if *x* is an IEEE NaN (Not A Number) object. float-trapping-nan-p returns true only if *x* is a trapping NaN. With either function, *x* must be a float.

2.1.3.2 Negative Zero

The IEEE float format provides for distinct positive and negative zeros. To test the sign on zero (or any other float), use the Common Lisp float-sign function. Negative zero prints as -0.0f0 or -0.0d0.

2.1.3.3 Denormalized Floats

CMUCL supports IEEE denormalized floats. Denormalized floats provide a mechanism for gradual underflow. The Common Lisp float-precision function returns the actual precision of a denormalized float, which will be less than float-digits. Note that in order to generate (or even print) denormalized floats, trapping must be disabled for the underflow exception (see section 2.1.3.4, page 6.) The Common Lisp least-positive-*format*-float constants are denormalized.

extensions:float-normalized-p *x* [*Function*]

This function returns true if *x* is a denormalized float. *x* must be a float.

2.1.3.4 Floating Point Exceptions

The IEEE floating point standard defines several exceptions that occur when the result of a floating point operation is unclear or undesirable. Exceptions can be ignored, in which case some default action is taken, such as returning a special value. When trapping is enabled for an exception, a error is signalled whenever that exception occurs. These are the possible floating point exceptions:

:underflow This exception occurs when the result of an operation is too small to be represented as a normalized float in its format. If trapping is enabled, the floating-point-underflow condition is signalled. Otherwise, the operation results in a denormalized float or zero.

:overflow This exception occurs when the result of an operation is too large to be represented as a float in its format. If trapping is enabled, the floating-point-overflow exception is signalled. Otherwise, the operation results in the appropriate infinity.

:inexact This exception occurs when the result of a floating point operation is not exact, i.e. the result was rounded. If trapping is enabled, the extensions:floating-point-inexact condition is signalled. Otherwise, the rounded result is returned.

:invalid This exception occurs when the result of an operation is ill-defined, such as (/ 0.0 0.0). If trapping is enabled, the extensions:floating-point-invalid condition is signalled. Otherwise, a quiet NaN is returned.

:divide-by-zero
This exception occurs when a float is divided by zero. If trapping is enabled, the divide-by-zero condition is signalled. Otherwise, the appropriate infinity is returned.

2.1.3.5 Floating Point Rounding Mode

IEEE floating point specifies four possible rounding modes:

:nearest
: In this mode, the inexact results are rounded to the nearer of the two possible result values. If the neither possibility is nearer, then the even alternative is chosen. This form of rounding is also called "round to even", and is the form of rounding specified for the Common Lisp round function.

:positive-infinity
: This mode rounds inexact results to the possible value closer to positive infinity. This is analogous to the Common Lisp ceiling function.

:negative-infinity
: This mode rounds inexact results to the possible value closer to negative infinity. This is analogous to the Common Lisp floor function.

:zero
: This mode rounds inexact results to the possible value closer to zero. This is analogous to the Common Lisp truncate function.

Warning: Although the rounding mode can be changed with set-floating-point-modes, use of any value other than the default (:nearest) can cause unusual behavior, since it will affect rounding done by Common Lisp system code as well as rounding in user code. In particular, the unary round function will stop doing round-to-nearest on floats, and instead do the selected form of rounding.

2.1.3.6 Accessing the Floating Point Modes

These functions can be used to modify or read the floating point modes:

extensions:set-floating-point-modes &key :traps :rounding-mode :fast-mode :accrued-exceptions :current-exceptions [*Function*]

extensions:get-floating-point-modes [*Function*]

The keyword arguments to set-floating-point-modes set various modes controlling how floating point arithmetic is done:

:traps
: A list of the exception conditions that should cause traps. Possible exceptions are :underflow, :overflow, :inexact, :invalid and :divide-by-zero. Initially all traps except :inexact are enabled. See section 2.1.3.4, page 6.

:rounding-mode
: The rounding mode to use when the result is not exact. Possible values are :nearest, :positive-infinity, :negative-infinity and :zero. Initially, the rounding mode is :nearest. See the warning in section 2.1.3.5 about use of other rounding modes.

:current-exceptions, :accrued-exceptions
: Lists of exception keywords used to set the exception flags. The *current-exceptions* are the exceptions for the previous operation, so setting it is not very useful. The *accrued-exceptions* are a cumulative record of the exceptions that occurred since the last time these flags were cleared. Specifying () will clear any accrued exceptions.

:fast-mode
: Set the hardware's "fast mode" flag, if any. When set, IEEE conformance or debuggability may be impaired. Some machines may not have this feature, in which case the value is always nil. No currently supported machines have a fast mode.

If a keyword argument is not supplied, then the associated state is not changed.

get-floating-point-modes returns a list representing the state of the floating point modes. The list is in the same format as the keyword arguments to set-floating-point-modes, so apply could be used with set-floating-point-modes to restore the modes in effect at the time of the call to get-floating-point-modes.

To make handling control of floating-point exceptions, the following macro is useful.

ext:with-float-traps-masked traps &body body [*Macro*]

body is executed with the selected floating-point exceptions given by traps masked out (disabled). traps should be a list of possible floating-point exceptions that should be ignored. Possible values are :underflow, :overflow, :inexact, :invalid and :divide-by-zero.

This is equivalent to saving the current traps from get-floating-point-modes, setting the floating-point modes to the desired exceptions, running the body, and restoring the saved floating-point modes. The advantage of this macro is that it causes less consing to occur.

Some points about the with-float-traps-masked:

- Two approaches are available for detecting FP exceptions:
  1. enabling the traps and handling the exceptions
  2. disabling the traps and either handling the return values or checking the accrued exceptions.

  Of these the latter is the most portable because on the alpha port it is not possible to enable some traps at run-time.
- To assist the checking of the exceptions within the body any accrued exceptions matching the given traps are cleared at the start of the body when the traps are masked.
- To allow the macros to be nested these accrued exceptions are restored at the end of the body to their values at the start of the body. Thus any exceptions that occurred within the body will not affect the accrued exceptions outside the macro.
- Note that only the given exceptions are restored at the end of the body so other exception will be visible in the accrued exceptions outside the body.
- On the x86, setting the accrued exceptions of an unmasked exception would cause a FP trap. The macro behaviour of restoring the accrued exceptions ensures than if an accrued exception is initially not flagged and occurs within the body it will be restored/cleared at the exit of the body and thus not cause a trap.
- On the x86, and, perhaps, the hppa, the FP exceptions may be delivered at the next FP instruction which requires a FP wait instruction (x86::float-wait) if using the lisp conditions to catch trap within a handler-bind. The handler-bind macro does the right thing and inserts a float-wait (at the end of its body on the x86). The masking and noting of exceptions is also safe here.
- The setting of the FP flags uses the (floating-point-modes) and the (set (floating-point-modes)...) VOPs. These VOPs blindly update the flags which may include other state. We assume this state hasn't changed in between getting and setting the state. For example, if you used the FP unit between the above calls, the state may be incorrectly restored! The with-float-traps-masked macro keeps the intervening code to a minimum and uses only integer operations.

2.1.4 Characters

CMUCL implements characters according to *Common Lisp: The Language II*. The main difference from the first version is that character bits and font have been eliminated, and the names of the types have been changed. base-character is the new equivalent of the old string-char. In this implementation, all characters are base characters (there are no extended characters.) Character codes range between 0 and 255, using the ASCII encoding. Table 2.1 on the following page shows characters recognized by CMUCL.

2.1.5 Array Initialization

If no :initial-value is specified, arrays are initialized to zero.

ASCII		Lisp	Alternatives		
Name	Code	Name			
nul	0	#\NULL	#\NUL		
bel	7	#\BELL			
bs	8	#\BACKSPACE	#\BS		
tab	9	#\TAB			
lf	10	#\NEWLINE	#\NL	#\LINEFEED	#\LF
ff	11	#\VT	#\PAGE	#\FORM	
cr	13	#\RETURN	#\CR		
esc	27	#\ESCAPE	#\ESC	#\ALTMODE	#\ALT
sp	32	#\SPACE	#\SP		
del	127	#\DELETE	#\RUBOUT		

Table 2.1: Characters recognized by CMUCL

2.2 Default Interrupts for Lisp

CMUCL has several interrupt handlers defined when it starts up, as follows:

SIGINT (↑c)
: 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 (↑L)
: 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 (↑z)
: 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.

For keyboard interrupt signals, the standard interrupt character is in parentheses. Your '`.login`' may set up different interrupt characters. 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. See section 6.8.1, page 105 to define your own signal handlers.

2.3 Packages

When CMUCL is first started up, the default package is the common-lisp-user package. The common-lisp-user package uses the common-lisp, extensions, and pcl packages. The symbols exported from these three packages can be referenced without package qualifiers. This section describes packages which have exported interfaces that may concern users. The numerous internal packages which implement parts of the system are not described here. Package nicknames are in parenthesis after the full name.

alien, c-call
: Export the features of the Alien foreign data structure facility (see section 8, page 115.)

pcl
: This package contains PCL (Portable CommonLoops), which is a portable implementation of CLOS (the Common Lisp Object System.) This implements most (but not all) of the features in the CLOS chapter of *Common Lisp: The Language II*.

debug
: The debug package contains the command-line oriented debugger. It exports utility various functions and switches.

debug-internals
: The debug-internals package exports the primitives used to write debuggers. See section 10, page 135.

extensions (ext)
: The extensions packages exports local extensions to Common Lisp that are documented in this manual. Examples include the save-lisp function and time parsing.

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.

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).

profile
: The profile package exports a simple run-time profiling facility (see section 5.14, page 95).

common-lisp (cl lisp)
: The common-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.

unix
: This package exports system call interfaces to Unix (see section 6, page 100).

system (sys)
: The system package contains functions and information necessary for system interfacing. This package is used by the lisp package and exports several symbols that are necessary to interface to system code.

xlib
: The xlib package contains the Common Lisp X interface (CLX) to the X11 protocol. This is mostly Lisp code with a couple of functions that are defined in C to connect to the server.

wire
: The wire package exports a remote procedure call facility (see section 9, page 129).

2.4 The Editor

The ed function invokes the Hemlock editor which is described in *Hemlock User's Manual* and *Hemlock Command Implementor's Manual*. Most users at CMU prefer to use Hemlock's slave Common Lisp mechanism which provides an interactive buffer for the read-eval-print loop and editor commands for evaluating and compiling text from a buffer into the slave Common Lisp. Since the editor runs in the Common Lisp, using slaves keeps users from trashing their editor by developing in the same Common Lisp with Hemlock.

2.5 Garbage Collection

CMUCL uses a stop-and-copy garbage collector that compacts the items in dynamic space every time it runs. Most users cause the system to garbage collect (GC) frequently, long before space is exhausted. With 16 or 24 megabytes of memory, causing GC's more frequently on less garbage allows the system to GC without much (if any) paging.

The following functions invoke the garbage collector or control whether automatic garbage collection is in effect:

extensions:gc [*Function*]

This function runs the garbage collector. If ext:*gc-verbose* is non-nil, then it invokes ext:*gc-notify-before* before GC'ing and ext:*gc-notify-after* afterwards.

extensions:gc-off [*Function*]

This function inhibits automatic garbage collection. After calling it, the system will not GC unless you call ext:gc or ext:gc-on.

extensions:gc-on [*Function*]

This function reinstates automatic garbage collection. If the system would have GC'ed while automatic GC was inhibited, then this will call ext:gc.

2.5.1 GC Parameters

The following variables control the behavior of the garbage collector:

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

CMUCL automatically GC's whenever the amount of memory allocated to dynamic objects exceeds the value of an internal variable. After each GC, the system sets this internal variable to the amount of dynamic space in use at that point plus the value of the variable ext:*bytes-consed-between-gcs*. The default value is 2000000.

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

This variable controls whether ext:gc invokes the functions in ext:*gc-notify-before* and ext:*gc-notify-after*. If *gc-verbose* is nil, ext:gc foregoes printing any messages. The default value is T.

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

This variable's value is a function that should notify the user that the system is about to GC. It takes one argument, the amount of dynamic space in use before the GC measured in bytes. 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*]

This variable's value is a function that should notify the user when a GC finishes. The function must take three arguments, 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. All values are byte quantities. 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 ext:*gc-notify-after* function. The system periodically checks whether this threshold has been exceeded, and only then does a garbage collection.

extensions:*gc-inhibit-hook* [*Variable*]

This variable's value is either a function of one argument or nil. When the system has triggered an automatic GC, if this variable is a function, then the system calls the function with the amount of dynamic space currently in use (measured in bytes). If the function returns nil, then the GC occurs; otherwise, the system inhibits automatic GC as if you had called ext:gc-off. The writer of this hook is responsible for knowing when automatic GC has been turned off and for calling or providing a way to call ext:gc-on. The default value of this variable is nil.

extensions:*before-gc-hooks* [*Variable*]
extensions:*after-gc-hooks* [*Variable*]

These variables' values are lists of functions to call before or after any GC occurs. The system provides these purely for side-effect, and the functions take no arguments.

2.5.2 Weak Pointers

A weak pointer provides a way to maintain a reference to an object without preventing an object from being garbage collected. If the garbage collector discovers that the only pointers to an object are weak pointers, then it breaks the weak pointers and deallocates the object.

extensions:make-weak-pointer *object* [*Function*]
extensions:weak-pointer-value *weak-pointer* [*Function*]

make-weak-pointer returns a weak pointer to an object. weak-pointer-value follows a weak pointer, returning the two values: the object pointed to (or nil if broken) and a boolean value which is true if the pointer has been broken.

2.5.3 Finalization

Finalization provides a "hook" that is triggered when the garbage collector reclaims an object. It is usually used to recover non-Lisp resources that were allocated to implement the finalized Lisp object. For example, when a unix file-descriptor stream is collected, finalization is used to close the underlying file descriptor.

extensions:finalize *object function* [*Function*]

This function registers *object* for finalization. *function* is called with no arguments when *object* is reclaimed. Normally *function* will be a closure over the underlying state that needs to be freed, e.g. the unix file descriptor in the fd-stream case. Note that *function* must not close over *object* itself, as this prevents the object from ever becoming garbage.

extensions:cancel-finalization *object* [*Function*]

This function cancel any finalization request for *object*.

2.6 Describe

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 function documentation, then it 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 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 indenting output.

A number of switches can be used to control describe's behavior.

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-print-level* [*Variable*]
extensions:*describe-print-length* [*Variable*]

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

2.7 The Inspector

CMUCL has both a graphical inspector that uses the X Window System, and a simple terminal-based inspector.

inspect &optional *object* [*Function*]

inspect calls the inspector on the optional argument *object*. If *object* is unsupplied, inspect immediately returns nil. Otherwise, the behavior of inspect depends on whether Lisp is running under X. When inspect is eventually exited, it returns some selected Lisp object.

2.7.1 The Graphical Interface

CMUCL has an interface to Motif which is functionally similar to CLM, but works better in CMUCL. This interface is documented in separate manuals *CMUCL Motif Toolkit* and *Design Notes on the Motif Toolkit*, which are distributed with CMUCL.

This motif interface has been used to write the inspector and graphical debugger. There is also a Lisp control panel with a simple file management facility, apropos and inspector dialogs, and controls for setting global options. See the interface and toolkit packages.

interface:lisp-control-panel [*Function*]

This function creates a control panel for the Lisp process.

interface:*interface-style* [*Variable*]

When the graphical interface is loaded, this variable controls whether it is used by inspect and the error system. If the value is :graphics (the default) and the DISPLAY environment variable is defined, the graphical inspector and debugger will be invoked by inspect or when an error is signalled. Possible values are :graphics and tty. If the value is :graphics, but there is no X display, then we quietly use the TTY interface.

2.7.2 The TTY Inspector

If X is unavailable, a terminal inspector is invoked. The TTY inspector is a crude interface to describe which allows objects to be traversed and maintains a history. This inspector prints information about and object and a numbered list of the components of the object. The command-line based interface is a normal read–eval–print loop, but an integer *n* descends into the *n*'th component of the current object, and symbols with these special names are interpreted as commands:

U Move back to the enclosing object. 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 Return the current object from inspect.

R Recompute object display, and print again. Useful if the object may have changed.

D Display again without recomputing.

H, ? Show help message.

2.8 Load

load *filename* &key :verbose :print :if-does-not-exist
:if-source-newer :contents [*Function*]

As in standard Common Lisp, this function loads a file containing source or object code into the running Lisp. Several CMU extensions have been made to load to conveniently support a variety of program file organizations. *filename* may be a wildcard pathname such as '`*.lisp`', in which case all matching files are loaded.

If *filename* has a pathname-type (or extension), then that exact file is loaded. If the file has no extension, then this tells load to use a heuristic to load the "right" file. The *load-source-types* and *load-object-types* variables below are used to determine the default source and object file types. If only the source or the object file exists (but not both), then that file is quietly loaded. Similarly, if both the source and object file exist, and the object file is newer than the source file, then the object file is loaded. The value of the *if-source-newer* argument is used to determine what action to take when both the source and object files exist, but the object file is out of date:

:load-object
: The object file is loaded even though the source file is newer.

: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.

This argument defaults to the value of ext:*load-if-source-newer* (initially :load-object.)

The *contents* argument can be used to override the heuristic (based on the file extension) that normally determines whether to load the file as a source file or an object file. If non-null, this argument must be either :source or :binary, which forces loading in source and binary mode, respectively. You really shouldn't ever need to use this argument.

extensions:*load-source-types* [*Variable*]
extensions:*load-object-types* [*Variable*]

These variables are lists of possible pathname-type values for source and object files to be passed to load. These variables are only used when the file passed to load has no type; in this case, the possible source and object types are used to default the type in order to determine the names of the source and object files.

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

This variable determines the default value of the *if-source-newer* argument to load. Its initial value is :load-object.

2.9 The Reader

extensions:*ignore-extra-close-parentheses* [*Variable*]

If this variable is t (the default), then the reader merely prints a warning when an extra close parenthesis is detected (instead of signalling an error.)

2.10 Stream Extensions

extensions:read-n-bytes *stream buffer start numbytes* &optional *eof-error-p* [*Function*]

On streams that support it, this function reads multiple bytes of data into a buffer. The buffer must be a simple-string or (simple-array (unsigned-byte 8) (*)). The argument *nbytes* specifies the desired number of bytes, and the return value is the number of bytes actually read.

- If *eof-error-p* is true, an end-of-file condition is signalled if end-of-file is encountered before *count* bytes have been read.
- If *eof-error-p* is false, read-n-bytes reads as much data is currently available (up to count bytes.) On pipes or similar devices, this function returns as soon as any data is available, even if the amount read is less than *count* and eof has not been hit. See also make-fd-stream (page 104).

2.11 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 :pty :input
:if-input-does-not-exist
:output :if-output-exists
:error :if-error-exists
:status-hook :before-execve [*Function*]

run-program runs *program* in a child process. *Program* should be a pathname or string naming the program. *Args* should be a list of strings which this passes to *program* as normal Unix parameters. For no arguments, specify *args* as nil. The value returned is either a process structure or nil. The process interface follows the description of run-program. If run-program fails to fork the child process, it returns nil.

Except for sharing file descriptors as explained in keyword argument descriptions, run-program closes all file descriptors in the child process before running the program. When you are done using a process, call process-close to reclaim system resources. You only need to do this when you supply :stream for one of :input, :output, or :error, or you supply :pty non-nil. You can call process-close regardless of whether you must to reclaim resources without penalty if you feel safer.

run-program accepts the following keyword arguments:

:env — This is an a-list mapping keywords and simple-strings. The default is ext:*environment-list*. If :env is specified, run-program 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.

:pty — This should be one of t, nil, or a stream. If specified non-nil, the subprocess executes under a Unix PTY. If specified as a stream, the system collects all output to this pty and writes it to this stream. If specified as t, the process-pty slot contains a stream from which you can read the program's output and to which you can write input for the program. The default is nil.

:input — This specifies how the program gets its input. If specified as a string, it is the name of a file that contains input for the child process. run-program opens the file as standard input. If specified as nil (the default), then standard input is the file '/dev/null'. If specified as t, the program uses the current standard input. This may cause some confusion if :wait is nil since two processes may use the terminal at the same time. If specified as :stream, then the process-input slot contains an output stream. Anything written to this stream goes to the program as input. :input may also be an input stream that already contains all the input for the process. In this case run-program reads all the input from this stream before returning, so this cannot be used to interact with the process.

:if-input-does-not-exist
: This specifies what to do if the input file does not exist. The following values are valid: nil (the default) causes run-program to return nil without doing anything; :create creates the named file; and :error signals an error.

:output
: This specifies what happens with the program's output. If specified as a pathname, it is the name of a file that contains output the program writes to its standard output. If specified as nil (the default), all output goes to '/dev/null'. If specified as t, the program writes to the Lisp process's standard output. This may cause confusion if :wait is nil since two processes may write to the terminal at the same time. If specified as :stream, then the process-output slot contains an input stream from which you can read the program's output.

:if-output-exists
: This specifies what to do if the output file already exists. The following values are valid: nil causes run-program to return nil without doing anything; :error (the default) signals an error; :supersede overwrites the current file; and :append appends all output to the file.

:error
: This is similar to :output, except the file becomes the program's standard error. Additionally, :error can be :output in which case the program's error output is routed to the same place specified for :output. If specified as :stream, the process-error contains a stream similar to the process-output slot when specifying the :output argument.

:if-error-exists
: This specifies what to do if the error output file already exists. It accepts the same values as :if-output-exists.

:status-hook
: This specifies a function to call whenever the process changes status. This is especially useful when specifying :wait as nil. The function takes the process as a required argument.

:before-execve
: This specifies a function to run in the child process before it becomes the program to run. This is useful for actions such as authenticating the child process without modifying the parent Lisp process.

2.11.1 Process Accessors

The following functions interface the process returned by run-program:

extensions:process-p *thing* [*Function*]

This function returns t if *thing* is a process. Otherwise it returns nil

extensions:process-pid *process* [*Function*]

This function returns the process ID, an integer, for the *process*.

extensions:process-status *process* [*Function*]

This function returns the current status of *process*, which is one of :running, :stopped, :exited, or :signaled.

extensions:process-exit-code *process* [*Function*]

This function returns either the exit code for *process*, if it is :exited, or the termination signal *process* if it is :signaled. The result is undefined for processes that are still alive.

extensions:process-core-dumped *process* [*Function*]

This function returns t if someone used a Unix signal to terminate the *process* and caused it to dump a Unix core image.

extensions:process-pty *process* [*Function*]

This function returns either the two-way stream connected to *process*'s Unix PTY connection or nil if there is none.

extensions:process-input *process* [*Function*]
extensions:process-output *process* [*Function*]
extensions:process-error *process* [*Function*]

If the corresponding stream was created, these functions return the input, output or error fd-stream. nil is returned if there is no stream.

extensions:process-status-hook *process* [*Function*]

This function returns the current function to call whenever *process*'s status changes. This function takes the *process* as a required argument. process-status-hook is setf'able.

extensions:process-plist *process* [*Function*]

This function returns annotations supplied by users, and it is setf'able. This is available solely for users to associate information with *process* without having to build a-lists or hash tables of process structures.

extensions:process-wait *process* &optional *check-for-stopped* [*Function*]

This function waits for *process* to finish. If *check-for-stopped* is non-nil, this also returns when *process* stops.

extensions:process-kill *process signal* &optional *whom* [*Function*]

This function sends the Unix *signal* to *process*. *Signal* should be the number of the signal or a keyword with the Unix name (for example, :sigsegv). *Whom* should be one of the following:

:pid This is the default, and it indicates sending the signal to *process* only.

:process-group
This indicates sending the signal to *process*'s group.

:pty-process-group
This indicates sending the signal to the process group currently in the foreground on the Unix PTY connected to *process*. This last option is useful if the running program is a shell, and you wish to signal the program running under the shell, not the shell itself. If process-pty of *process* is nil, using this option is an error.

extensions:process-alive-p *process* [*Function*]

This function returns t if *process*'s status is either :running or :stopped.

extensions:process-close *process* [*Function*]

This function closes all the streams associated with *process*. When you are done using a process, call this to reclaim system resources.

2.12 Saving a Core Image

A mechanism has been provided to save a running Lisp core image and to later restore it. This is convenient if you don't want to load several files into a Lisp when you first start it up. The main problem is the large size of each saved Lisp image, typically at least 20 megabytes.

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

The save-lisp function saves the state of the currently running Lisp core image in *file*. The keyword arguments have the following meaning:

:purify
: If non-nil (the default), the core image is purified before it is saved (see purify (page 18).) 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.

:root-structures
: This should be a list of the main entry points in any newly loaded systems. This need not be supplied, but locality and/or GC performance will be better if they are. Meaningless if :purify is nil. See purify (page 18).

:init-function
: This is the function that starts running when the created core file is resumed. The default function simply invokes the top level read-eval-print loop. If the function returns the lisp will exit.

:load-init-file
: If non-NIL, then load an init file; either the one specified on the command line or "'`init.`'*fasl-type*", or, if "'`init.`'*fasl-type*" 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.

:site-init
: If non-NIL, the name of the site init file to quietly load. The default is '`library:site-init`'. No error is signalled if the file does not exist.

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

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

To resume a saved file, type:

```
lisp -core file
```

extensions:purify *file* &key :root-structures :environment-name [*Function*]

This function optimizes garbage collection by moving all currently live objects into non-collected storage. Once statically allocated, the objects can never be reclaimed, even if all pointers to them are dropped. This function should generally be called after a large system has been loaded and initialized.

:root-structures
: is an optional list of objects which should be copied first to maximize locality. 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.

: *defstruct* structures defined with the (:pure `t`) option are moved into read-only storage, further reducing GC cost. List and vector slots of pure structures are also moved into read-only storage.

:environment-name
: is gratuitous documentation for the compacted version of the current global environment (as seen in c::*info-environment*.) If nil is supplied, then environment compaction is inhibited.

2.13 Pathnames

In Common Lisp quite a few aspects of pathname semantics are left to the implementation.

2.13.1 Unix Pathnames

Unix pathnames are always parsed with a unix-host object as the host and nil as the device. The last two dots (.) in the namestring mark the type and version, however if the first character is a dot, it is considered part of the name. If the last character is a dot, then the pathname has the empty-string as its type. The type defaults to nil and the version defaults to :newest.

```
(defun parse (x)
  (values (pathname-name x) (pathname-type x) (pathname-version x)))

(parse "foo") ⇒ "foo", NIL, :NEWEST
(parse "foo.bar") ⇒ "foo", "bar", :NEWEST
(parse ".foo") ⇒ ".foo", NIL, :NEWEST
(parse ".foo.bar") ⇒ ".foo", "bar", :NEWEST
(parse "..") ⇒ ".", "", :NEWEST
(parse "foo.") ⇒ "foo", "", :NEWEST
(parse "foo.bar.1") ⇒ "foo", "bar", 1
(parse "foo.bar.baz") ⇒ "foo.bar", "baz", :NEWEST
```

The directory of pathnames beginning with a slash (or a search-list, see section 2.13.4, page 20) is starts :absolute, others start with :relative. The .. directory is parsed as :up; there is no namestring for :back:

```
(pathname-directory "/usr/foo/bar.baz") ⇒ (:ABSOLUTE "usr" "foo")
(pathname-directory "../foo/bar.baz") ⇒ (:RELATIVE :UP "foo")
```

2.13.2 Wildcard Pathnames

Wildcards are supported in Unix pathnames. If '*' is specified for a part of a pathname, that is parsed as :wild. '**' can be used as a directory name to indicate :wild-inferiors. Filesystem operations treat :wild-inferiors the same as :wild, but pathname pattern matching (e.g. for logical pathname translation, see section 2.13.3, page 19) matches any number of directory parts with '**' (see see section 2.14.1, page 21.)

'*' embedded in a pathname part matches any number of characters. Similarly, '?' matches exactly one character, and '[a,b]' matches the characters 'a' or 'b'. These pathname parts are parsed as pattern objects.

Backslash can be used as an escape character in namestring parsing to prevent the next character from being treated as a wildcard. Note that if typed in a string constant, the backslash must be doubled, since the string reader also uses backslash as a quote:

```
(pathname-name "foo\\*bar") => "foo*bar"
```

2.13.3 Logical Pathnames

If a namestring begins with the name of a defined logical pathname host followed by a colon, then it will be parsed as a logical pathname. Both '*' and '**' wildcards are implemented. load-logical-pathname-translations on *name* looks for a logical host definition file in '`library:`*name*`.translations`'. Note that '`library:`' designates the search list (see section 2.13.4, page 20) initialized to the CMUCL '`lib/`' directory, not a logical pathname. The format of the file is a single list of two-lists of the from and to patterns:

```
(("foo;*.text" "/usr/ram/foo/*.txt")
 ("foo;*.lisp" "/usr/ram/foo/*.l"))
```

2.13.4 Search Lists

Search lists are an extension to Common Lisp pathnames. They serve a function somewhat similar to Common Lisp logical pathnames, but work more like Unix PATH variables. Search lists are used for two purposes:

- They provide a convenient shorthand for commonly used directory names, and
- They allow the abstract (directory structure independent) specification of file locations in program pathname constants (similar to logical pathnames.)

Each search list has an associated list of directories (represented as pathnames with no name or type component.) The namestring for any relative pathname may be prefixed with "*slist*:", indicating that the pathname is relative to the search list *slist* (instead of to the current working directory.) Once qualified with a search list, the pathname is no longer considered to be relative.

When a search list qualified pathname is passed to a file-system operation such as open, load or truename, each directory in the search list is successively used as the root of the pathname until the file is located. When a file is written to a search list directory, the file is always written to the first directory in the list.

2.13.5 Predefined Search-Lists

These search-lists are initialized from the Unix environment or when Lisp was built:

default: The current directory at startup.

home: The user's home directory.

library: The CMUCL '`lib/`' directory (CMUCLLIB environment variable.)

path: The Unix command path (PATH environment variable.)

target: The root of the tree where CMUCL was compiled.

It can be useful to redefine these search-lists, for example, '`library:`' can be augmented to allow logical pathname translations to be located, and '`target:`' can be redefined to point to where CMUCL system sources are locally installed.

2.13.6 Search-List Operations

These operations define and access search-list definitions. A search-list name may be parsed into a pathname before the search-list is actually defined, but the search-list must be defined before it can actually be used in a filesystem operation.

extensions:search-list *name* [*Function*]

This function returns the list of directories associated with the search list *name*. If *name* is not a defined search list, then an error is signaled. When set with setf, the list of directories is changed to the new value. If the new value is just a namestring or pathname, then it is interpreted as a one-element list. Note that (unlike Unix pathnames), search list names are case-insensitive.

extensions:search-list-defined-p *name* [*Function*]
extensions:clear-search-list *name* [*Function*]

search-list-defined-p returns t if *name* is a defined search list name, nil otherwise. clear-search-list make the search list *name* undefined.

extensions:enumerate-search-list (*var pathname* {*result*}) {*form*}* [*Macro*]

This macro provides an interface to search list resolution. The body *forms* are executed with *var* bound to each successive possible expansion for *name*. If *name* does not contain a search-list, then the body is executed exactly once. Everything is wrapped in a block named nil, so return can be used to terminate early. The *result* form (default nil) is evaluated to determine the result of the iteration.

2.13.7 Search List Example

The search list code: can be defined 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. For example, calling ext:search-list on code: as follows:

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

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

2.14 Filesystem Operations

CMUCL provides a number of extensions and optional features beyond those require by Common Lisp.

2.14.1 Wildcard Matching

Unix filesystem operations such as open will accept wildcard pathnames that match a single file (of course, directory allows any number of matches.) Filesystem operations treat :wild-inferiors the same as :wild.

directory *wildname* &key :all :check-for-subdirs :truenamep :follow-links [*Function*]

The keyword arguments to this Common Lisp function are a CMU extension. The arguments (all default to t) have the following functions:

:all
: Include files beginning with dot such as '.login', similar to "ls -a".

:check-for-subdirs
: Test whether files are directories, similar to "ls -F".

:truenamep
: Call truename on each file, which expands out all symbolic links. Note that this option can easily result in pathnames being returned which have a different directory from the one in the *wildname* argument.

:follow-links
: Follow symbolic links when searching for matching directories.

extensions:print-directory *wildname* &optional *stream* &key :all :verbose :return-list [*Function*]

Print a directory of *wildname* listing to *stream* (default *standard-output*.) :all and :verbose both default to nil and correspond to the "-a" and "-l" options of 'ls'. Normally this function returns nil, but if :return-list is true, a list of the matched pathnames are returned.

2.14.2 File Name Completion

extensions:complete-file *pathname* &key :defaults :ignore-types [*Function*]

Attempt to complete a file name to the longest unambiguous prefix. If supplied, directory from :defaults is used as the "working directory" when doing completion. :ignore-types is a list of strings of the pathname types (a.k.a. extensions) that should be disregarded as possible matches (binary file names, etc.)

extensions:ambiguous-files *pathname* &optional *defaults* [*Function*]

Return a list of pathnames for all the possible completions of *pathname* with respect to *defaults*.

2.14.3 Miscellaneous Filesystem Operations

extensions:default-directory [*Function*]

Return the current working directory as a pathname. If set with setf, set the working directory.

extensions:file-writable *name* [*Function*]

This function accepts a pathname and returns t if the current process can write it, and nil otherwise.

extensions:unix-namestring *pathname* &optional *for-input* [*Function*]

This function converts *pathname* into a string that can be used with UNIX system calls. Search-lists and wildcards are expanded. *for-input* controls the treatment of search-lists: when true (the default) and the file exists anywhere on the search-list, then that absolute pathname is returned; otherwise the first element of the search-list is used as the directory.

2.15 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 :default-month :default-year :default-zone :default-weekday [*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 :print-meridian
:print-timezone :print-weekday [*Function*]

extensions:format-decoded-time *dest seconds minutes hours day month year*
&key :timezone
:style :date-first
:print-seconds :print-meridian
:print-timezone :print-weekday [*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. :timezone defaults to the current time zone.

:style specifies the style to use in formatting 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-meridian
: 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-weekday
: if non-nil (default) will format the weekday as part of date. Otherwise, the weekday will be omitted.

2.16 Random Number Generation

Common Lisp includes a random number generator as a standard part of the language; however, the implementation of the generator is not specified. Two random number generators are available in CMUCL, depending on the version.

2.16.1 Original Generator

The default random number generator uses a lagged Fibonacci generator given by

$$z[i] = z[i - 24] - z[i - 55] \bmod 536870908$$

where $z[i]$ is the i'th random number. This generator produces small integer-valued numbers. For larger integer, the small random integers are concatenated to produce larger integers. For floating-point numbers, the bits from this generator are used as the bits of the floating-point significand.

2.16.2 New Generator

In some versions of CMUCL, the original generator above has been replaced with a subtract-with-borrow generator combined with a Weyl generator.[1] The reason for the change was to use a documented generator which has passed tests for randomness.

The subtract-with-borrow generator is described by the following equation

$$z[i] = z[i + 20] - z[i + 5] - b$$

where $z[i]$ is the i'th random number, which is a double-float. All of the indices in this equation are interpreted modulo 32. The quantity b is carried over from the previous iteration and is either 0 or double-float-epsilon. If $z[i]$ is positive, b is set to zero. Otherwise, b is set to double-float-epsilon.

To increase the randomness of this generator, this generator is combined with a Weyl generator defined by

$$x[i] = x[i - 1] - y \bmod 1,$$

where $y = 7097293079245107 \times 2^{-53}$. Thus, the resulting random number $r[i]$ is

$$r[i] = (z[i] - x[i]) \bmod 1$$

This generator has been tested by Peter VanEynde using Marsaglia's diehard test suite for random number generators; this generator passes the test suite.

This generator is designed for generating floating-point random numbers. To obtain integers, the bits from the significand of the floating-point number are used as the bits of the integer. As many floating-point numbers as needed are generated to obtain the desired number of bits in the random integer.

For floating-point numbers, this generator can by significantly faster than the original generator.

2.17 Lisp Library

The CMUCL project maintains a collection of useful or interesting programs written by users of our system. The library is in '`lib/contrib/`'. Two files there that users should read are:

CATALOG.TXT
: This file contains a page for each entry in the library. It contains information such as the author, portability or dependency issues, how to load the entry, etc.

READ-ME.TXT
: This file describes the library's organization and all the possible pieces of information an entry's catalog description could contain.

Hemlock has a command Library Entry that displays a list of the current library entries in an editor buffer. There are mode specific commands that display catalog descriptions and load entries. This is a simple and convenient way to browse the library.

[1] The generator described here is available if the feature :new-random is available.

Chapter 3

The Debugger

by Robert MacLachlan

3.1 Debugger Introduction

The CMUCL debugger is unique in its level of support for source-level debugging of compiled code. Although some other debuggers allow access of variables by name, this seems to be the first Common Lisp debugger that:

- Tells you when a variable doesn't have a value because it hasn't been initialized yet or has already been deallocated, or
- Can display the precise source location corresponding to a code location in the debugged program.

These features allow the debugging of compiled code to be made almost indistinguishable from interpreted code debugging.

The debugger is an interactive command loop that allows a user to examine the function call stack. The debugger is invoked when:

- A serious-condition is signaled, and it is not handled, or
- error is called, and the condition it signals is not handled, or
- The debugger is explicitly invoked with the Common Lisp break or debug functions.

Note: there are two debugger interfaces in CMUCL: the TTY debugger (described below) and the Motif debugger. Since the difference is only in the user interface, much of this chapter also applies to the Motif version. See section 2.7.1, page 13 for a very brief discussion of the graphical interface.

When you enter the TTY debugger, it looks something like this:

```
Error in function CAR.
Wrong type argument, 3, should have been of type LIST.

Restarts:
  0: Return to Top-Level.

Debug  (type H for help)

(CAR 3)
0]
```

The first group of lines describe what the error was that put us in the debugger. In this case car was called on 3. After Restarts: is a list of all the ways that we can restart execution after this error. In this case, the only option is to return to top-level. After printing its banner, the debugger prints the current frame and the debugger prompt.

3.2 The Command Loop

The debugger is an interactive read-eval-print loop much like the normal top-level, but some symbols are interpreted as debugger commands instead of being evaluated. A debugger command starts with the symbol name of the command, possibly followed by some arguments on the same line. Some commands prompt for additional input. Debugger commands can be abbreviated by any unambiguous prefix: help can be typed as h, he, etc. For convenience, some commands have ambiguous one-letter abbreviations: f for frame.

The package is not significant in debugger commands; any symbol with the name of a debugger command will work. If you want to show the value of a variable that happens also to be the name of a debugger command, you can use the list-locals command or the debug:var function, or you can wrap the variable in a progn to hide it from the command loop.

The debugger prompt is "*frame*]", where *frame* is the number of the current frame. Frames are numbered starting from zero at the top (most recent call), increasing down to the bottom. The current frame is the frame that commands refer to. The current frame also provides the lexical environment for evaluation of non-command forms.

The debugger evaluates forms in the lexical environment of the functions being debugged. The debugger can only access variables. You can't go or return-from into a function, and you can't call local functions. Special variable references are evaluated with their current value (the innermost binding around the debugger invocation)—you don't get the value that the special had in the current frame. See section 3.4, page 29 for more information on debugger variable access.

3.3 Stack Frames

A stack frame is the run-time representation of a call to a function; the frame stores the state that a function needs to remember what it is doing. Frames have:

- Variables (see section 3.4, page 29), which are the values being operated on, and
- Arguments to the call (which are really just particularly interesting variables), and
- A current location (see section 3.5, page 30), which is the place in the program where the function was running when it stopped to call another function, or because of an interrupt or error.

3.3.1 Stack Motion

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:

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

down Move down to the next lower frame.

top Move to the highest frame.

bottom Move to the lowest frame.

frame [*n*] Move to the frame with the specified number. Prompts for the number if not supplied.

3.3.2 How Arguments are Printed

A frame is printed to look like a function call, but with the actual argument values in the argument positions. So the frame for this call in the source:

```
(myfun (+ 3 4) 'a)
```

would look like this:

```
(MYFUN 7 A)
```

All keyword and optional arguments are displayed with their actual values; if the corresponding argument was not supplied, the value will be the default. So this call:

```
(subseq "foo" 1)
```

would look like this:

```
(SUBSEQ "foo" 1 3)
```

And this call:

```
(string-upcase "test case")
```

would look like this:

```
(STRING-UPCASE "test case" :START 0 :END NIL)
```

The arguments to a function call are displayed by accessing the argument variables. Although those variables are initialized to the actual argument values, they can be set inside the function; in this case the new value will be displayed.

&rest arguments are handled somewhat differently. The value of the rest argument variable is displayed as the spread-out arguments to the call, so:

```
(format t "~A is a ~A." "This" 'test)
```

would look like this:

```
(FORMAT T "~A is a ~A." "This" 'TEST)
```

Rest arguments cause an exception to the normal display of keyword arguments in functions that have both &rest and &key arguments. In this case, the keyword argument variables are not displayed at all; the rest arg is displayed instead. So for these functions, only the keywords actually supplied will be shown, and the values displayed will be the argument values, not values of the (possibly modified) variables.

If the variable for an argument is never referenced by the function, it will be deleted. The variable value is then unavailable, so the debugger prints <unused-arg> instead of the value. Similarly, if for any of a number of reasons (described in more detail in section 3.4) the value of the variable is unavailable or not known to be available, then <unavailable-arg> will be printed instead of the argument value.

Printing of argument values is controlled by *debug-print-level* and *debug-print-length* (page 37).

3.3.3 Function Names

If a function is defined by defun, labels, or flet, then the debugger will print the actual function name after the open parenthesis, like:

```
(STRING-UPCASE "test case" :START 0 :END NIL)
((SETF AREF) #\a "for" 1)
```

Otherwise, the function name is a string, and will be printed in quotes:

```
("DEFUN MYFUN" BAR)
("DEFMACRO DO" (DO ((I 0 (1+ I))) ((= I 13))) NIL)
("SETQ *GC-NOTIFY-BEFORE*")
```

This string name is derived from the def*mumble* form that encloses or expanded into the lambda, or the outermost enclosing form if there is no def*mumble*.

3.3.4 Funny Frames

Sometimes the evaluator introduces new functions that are used to implement a user function, but are not directly specified in the source. The main place this is done is for checking argument type and syntax. Usually these functions do their thing and then go away, and thus are not seen on the stack in the debugger. But when you get some sort of error during lambda-list processing, you end up in the debugger on one of these funny frames.

These funny frames are flagged by printing "[*keyword*]" after the parentheses. For example, this call:

```
(car 'a 'b)
```

will look like this:

```
(CAR 2 A) [:EXTERNAL]
```

And this call:

```
(string-upcase "test case" :end)
```

would look like this:

```
("DEFUN STRING-UPCASE" "test case" 335544424 1) [:OPTIONAL]
```

As you can see, these frames have only a vague resemblance to the original call. Fortunately, the error message displayed when you enter the debugger will usually tell you what problem is (in these cases, too many arguments and odd keyword arguments.) Also, if you go down the stack to the frame for the calling function, you can display the original source (see section 3.5, page 30.)

With recursive or block compiled functions (see section 5.7, page 75), an :EXTERNAL frame may appear before the frame representing the first call to the recursive function or entry to the compiled block. This is a consequence of the way the compiler does block compilation: there is nothing odd with your program. You will also see :CLEANUP frames during the execution of unwind-protect cleanup code. Note that inline expansion and open-coding affect what frames are present in the debugger, see sections 3.6 and 4.8.

3.3.5 Debug Tail Recursion

Both the compiler and the interpreter are "properly tail recursive." If a function call is in a tail-recursive position, the stack frame will be deallocated *at the time of the call*, rather than after the call returns. Consider this backtrace:

```
(BAR ...)
(FOO ...)
```

Because of tail recursion, it is not necessarily the case that FOO directly called BAR. It may be that FOO called some other function FOO2 which then called BAR tail-recursively, as in this example:

```
(defun foo ()
  ...
  (foo2 ...)
  ...)

(defun foo2 (...)
  ...
  (bar ...))

(defun bar (...)
  ...)
```

Usually the elimination of tail-recursive frames makes debugging more pleasant, since theses frames are mostly uninformative. If there is any doubt about how one function called another, it can usually be eliminated by finding the source location in the calling frame (section 3.5.)

For a more thorough discussion of tail recursion, see section 5.5, page 71.

3.3.6 Unknown Locations and Interrupts

The debugger operates using special debugging information attached to the compiled code. This debug information tells the debugger what it needs to know about the locations in the code where the debugger can be invoked. If the debugger somehow encounters a location not described in the debug information, then it is said to be *unknown*. If the code location for a frame is unknown, then some variables may be inaccessible, and the source location cannot be precisely displayed.

There are three reasons why a code location could be unknown:

- There is inadequate debug information due to the value of the debug optimization quality. See section 3.6, page 32.
- The debugger was entered because of an interrupt such as ^C.
- A hardware error such as "bus error" occurred in code that was compiled unsafely due to the value of the safety optimization quality. See section 4.7.1, page 50.

In the last two cases, the values of argument variables are accessible, but may be incorrect. See section 3.4.1, page 30 for more details on when variable values are accessible.

It is possible for an interrupt to happen when a function call or return is in progress. The debugger may then flame out with some obscure error or insist that the bottom of the stack has been reached, when the real problem is that the current stack frame can't be located. If this happens, return from the interrupt and try again.

When running interpreted code, all locations should be known. However, an interrupt might catch some subfunction of the interpreter at an unknown location. In this case, you should be able to go up the stack a frame or two and reach an interpreted frame which can be debugged.

3.4 Variable Access

There are three ways to access the current frame's local variables in the debugger. The simplest is to type the variable's name into the debugger's read-eval-print loop. The debugger will evaluate the variable reference as though it had appeared inside that frame.

The debugger doesn't really understand lexical scoping; it has just one namespace for all the variables in a function. If a symbol is the name of multiple variables in the same function, then the reference appears ambiguous, even though lexical scoping specifies which value is visible at any given source location. If the scopes of the two variables are not nested, then the debugger can resolve the ambiguity by observing that only one variable is accessible.

When there are ambiguous variables, the evaluator assigns each one a small integer identifier. The debug:var function and the list-locals command use this identifier to distinguish between ambiguous variables:

list-locals {*prefix*}
: This command prints the name and value of all variables in the current frame whose name has the specified *prefix*. *prefix* may be a string or a symbol. If no *prefix* is given, then all available variables are printed. If a variable has a potentially ambiguous name, then the name is printed with a "#*identifier*" suffix, where *identifier* is the small integer used to make the name unique.

debug:var *name* &optional *identifier* [*Function*]

This function returns the value of the variable in the current frame with the specified *name*. If supplied, *identifier* determines which value to return when there are ambiguous variables.

When *name* is a symbol, it is interpreted as the symbol name of the variable, i.e. the package is significant. If *name* is an uninterned symbol (gensym), then return the value of the uninterned variable with the same name. If *name* is a string, debug:var interprets it as the prefix of a variable name, and must unambiguously complete to the name of a valid variable.

This function is useful mainly for accessing the value of uninterned or ambiguous variables, since most variables can be evaluated directly.

3.4.1 Variable Value Availability

The value of a variable may be unavailable to the debugger in portions of the program where Common Lisp says that the variable is defined. If a variable value is not available, the debugger will not let you read or write that variable. With one exception, the debugger will never display an incorrect value for a variable. Rather than displaying incorrect values, the debugger tells you the value is unavailable.

The one exception is this: if you interrupt (e.g., with ^C) or if there is an unexpected hardware error such as "bus error" (which should only happen in unsafe code), then the values displayed for arguments to the interrupted frame might be incorrect.[1] This exception applies only to the interrupted frame: any frame farther down the stack will be fine.

The value of a variable may be unavailable for these reasons:

- The value of the debug optimization quality may have omitted debug information needed to determine whether the variable is available. Unless a variable is an argument, its value will only be available when debug is at least 2.
- The compiler did lifetime analysis and determined that the value was no longer needed, even though its scope had not been exited. Lifetime analysis is inhibited when the debug optimization quality is 3.
- The variable's name is an uninterned symbol (gensym). To save space, the compiler only dumps debug information about uninterned variables when the debug optimization quality is 3.
- The frame's location is unknown (see section 3.3.6, page 29) because the debugger was entered due to an interrupt or unexpected hardware error. Under these conditions the values of arguments will be available, but might be incorrect. This is the exception above.
- The variable was optimized out of existence. Variables with no reads are always optimized away, even in the interpreter. The degree to which the compiler deletes variables will depend on the value of the compile-speed optimization quality, but most source-level optimizations are done under all compilation policies.

Since it is especially useful to be able to get the arguments to a function, argument variables are treated specially when the speed optimization quality is less than 3 and the debug quality is at least 1. With this compilation policy, the values of argument variables are almost always available everywhere in the function, even at unknown locations. For non-argument variables, debug must be at least 2 for values to be available, and even then, values are only available at known locations.

3.4.2 Note On Lexical Variable Access

When the debugger command loop establishes variable bindings for available variables, these variable bindings have lexical scope and dynamic extent.[2] You can close over them, but such closures can't be used as upward funargs.

You can also set local variables using setq, but if the variable was closed over in the original source and never set, then setting the variable in the debugger may not change the value in all the functions the variable is defined in. Another risk of setting variables is that you may assign a value of a type that the compiler proved the variable could never take on. This may result in bad things happening.

3.5 Source Location Printing

One of CMUCL's unique capabilities is source level debugging of compiled code. These commands display the source location for the current frame:

source {*context*}
: This command displays the file that the current frame's function was defined from (if it was defined from a file), and then the source form responsible for generating the code that the current frame was

[1] Since the location of an interrupt or hardware error will always be an unknown location (see section 3.3.6, page 29), non-argument variable values will never be available in the interrupted frame.

[2] The variable bindings are actually created using the Common Lisp symbol-macro-let special form.

executing. If *context* is specified, then it is an integer specifying the number of enclosing levels of list structure to print.

vsource {*context*}
: This command is identical to source, except that it uses the global values of *print-level* and *print-length* instead of the debugger printing control variables *debug-print-level* and *debug-print-length*.

The source form for a location in the code is the innermost list present in the original source that encloses the form responsible for generating that code. If the actual source form is not a list, then some enclosing list will be printed. For example, if the source form was a reference to the variable *some-random-special*, then the innermost enclosing evaluated form will be printed. Here are some possible enclosing forms:

```
(let ((a *some-random-special*))
  ...)

(+ *some-random-special* ...)
```

If the code at a location was generated from the expansion of a macro or a source-level compiler optimization, then the form in the original source that expanded into that code will be printed. Suppose the file '/usr/me/mystuff.lisp' looked like this:

```
(defmacro mymac ()
  '(myfun))

(defun foo ()
  (mymac)
  ...)
```

If foo has called myfun, and is waiting for it to return, then the source command would print:

```
; File: /usr/me/mystuff.lisp

(MYMAC)
```

Note that the macro use was printed, not the actual function call form, (myfun).

If enclosing source is printed by giving an argument to source or vsource, then the actual source form is marked by wrapping it in a list whose first element is #:***HERE***. In the previous example, source 1 would print:

```
; File: /usr/me/mystuff.lisp

(DEFUN FOO ()
  (#:***HERE***
   (MYMAC))
  ...)
```

3.5.1 How the Source is Found

If the code was defined from Common Lisp by compile or eval, then the source can always be reliably located. If the code was defined from a fasl file created by compile-file, then the debugger gets the source forms it prints by reading them from the original source file. This is a potential problem, since the source file might have moved or changed since the time it was compiled.

The source file is opened using the truename of the source file pathname originally given to the compiler. This is an absolute pathname with all logical names and symbolic links expanded. If the file can't be located using this name, then the debugger gives up and signals an error.

If the source file can be found, but has been modified since the time it was compiled, the debugger prints this warning:

```
; File has been modified since compilation:
;   filename
; Using form offset instead of character position.
```

where *filename* is the name of the source file. It then proceeds using a robust but not foolproof heuristic for locating the source. This heuristic works if:

- No top-level forms before the top-level form containing the source have been added or deleted, and
- The top-level form containing the source has not been modified much. (More precisely, none of the list forms beginning before the source form have been added or deleted.)

If the heuristic doesn't work, the displayed source will be wrong, but will probably be near the actual source. If the "shape" of the top-level form in the source file is too different from the original form, then an error will be signaled. When the heuristic is used, the the source location commands are noticeably slowed.

Source location printing can also be confused if (after the source was compiled) a read-macro you used in the code was redefined to expand into something different, or if a read-macro ever returns the same eq list twice. If you don't define read macros and don't use ## in perverted ways, you don't need to worry about this.

3.5.2 Source Location Availability

Source location information is only available when the debug optimization quality is at least 2. If source location information is unavailable, the source commands will give an error message.

If source location information is available, but the source location is unknown because of an interrupt or unexpected hardware error (see section 3.3.6, page 29), then the command will print:

```
Unknown location: using block start.
```

and then proceed to print the source location for the start of the *basic block* enclosing the code location. It's a bit complicated to explain exactly what a basic block is, but here are some properties of the block start location:

- The block start location may be the same as the true location.
- The block start location will never be later in the the program's flow of control than the true location.
- No conditional control structures (such as if, cond, or) will intervene between the block start and the true location (but note that some conditionals present in the original source could be optimized away.) Function calls *do not* end basic blocks.
- The head of a loop will be the start of a block.
- The programming language concept of "block structure" and the Common Lisp block special form are totally unrelated to the compiler's basic block.

In other words, the true location lies between the printed location and the next conditional (but watch out because the compiler may have changed the program on you.)

3.6 Compiler Policy Control

The compilation policy specified by optimize declarations affects the behavior seen in the debugger. The debug quality directly affects the debugger by controlling the amount of debugger information dumped. Other optimization qualities have indirect but observable effects due to changes in the way compilation is done.

Unlike the other optimization qualities (which are compared in relative value to evaluate tradeoffs), the debug optimization quality is directly translated to a level of debug information. This absolute interpretation allows the user to count on a particular amount of debug information being available even when the values of the other qualities are changed during compilation. These are the levels of debug information that correspond to the values of the debug quality:

0 Only the function name and enough information to allow the stack to be parsed.

> 0 Any level greater than 0 gives level 0 plus all argument variables. Values will only be accessible if the argument variable is never set and speed is not 3. CMUCL allows any real value for optimization qualities. It may be useful to specify 0.5 to get backtrace argument display without argument documentation.

1 Level 1 provides argument documentation (printed arglists) and derived argument/result type information. This makes describe more informative, and allows the compiler to do compile-time argument count and type checking for any calls compiled at run-time.

2 Level 1 plus all interned local variables, source location information, and lifetime information that tells the debugger when arguments are available (even when speed is 3 or the argument is set.) This is the default.

3 Level 2 plus all uninterned variables. In addition, lifetime analysis is disabled (even when speed is 3), ensuring that all variable values are available at any known location within the scope of the binding. This has a speed penalty in addition to the obvious space penalty.

As you can see, if the speed quality is 3, debugger performance is degraded. This effect comes from the elimination of argument variable special-casing (see section 3.4.1, page 30.) Some degree of speed/debuggability tradeoff is unavoidable, but the effect is not too drastic when debug is at least 2.

In addition to inline and notinline declarations, the relative values of the speed and space qualities also change whether functions are inline expanded (see section 5.8, page 79.) If a function is inline expanded, then there will be no frame to represent the call, and the arguments will be treated like any other local variable. Functions may also be "semi-inline", in which case there is a frame to represent the call, but the call is to an optimized local version of the function, not to the original function.

3.7 Exiting Commands

These commands get you out of the debugger.

quit
: Throw to top level.

restart {*n*}
: Invokes the *n*th restart case as displayed by the error command. If *n* is not specified, the available restart cases are reported.

go
: Calls continue on the condition given to debug. If there is no restart case named *continue*, then an error is signaled.

abort
: Calls abort on the condition given to debug. This is useful for popping debug command loop levels or aborting to top level, as the case may be.

3.8 Information Commands

Most of these commands print information about the current frame or function, but a few show general information.

help, ?
: Displays a synopsis of debugger commands.

describe
: Calls describe on the current function, displays number of local variables, and indicates whether the function is compiled or interpreted.

print
: Displays the current function call as it would be displayed by moving to this frame.

vprint (or pp) {*verbosity*}
: Displays the current function call using *print-level* and *print-length* instead of *debug-print-level* and *debug-print-length*. *verbosity* is a small integer (default 2) that controls other dimensions of verbosity.

error
: Prints the condition given to invoke-debugger and the active proceed cases.

backtrace {*n*}
: Displays all the 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*.

3.9 Breakpoint Commands

CMUCL supports setting of breakpoints inside compiled functions and stepping of compiled code. Breakpoints can only be set at at known locations (see section 3.3.6, page 29), so these commands are largely useless unless the debug optimize quality is at least 2 (see section 3.6, page 32). These commands manipulate breakpoints:

breakpoint *location* {*option value*}*
: Set a breakpoint in some function. *location* may be an integer code location number (as displayed by list-locations) or a keyword. The keyword can be used to indicate setting a breakpoint at the function start (:start, :s) or function end (:end, :e). The breakpoint command has :condition, :break, :print and :function options which work similarly to the trace options.

list-locations (or ll) {*function*}
: List all the code locations in the current frame's function, or in *function* if it is supplied. The display format is the code location number, a colon and then the source form for that location:

```
3: (1- N)
```

If consecutive locations have the same source, then a numeric range like 3-5: will be printed. For example, a default function call has a known location both immediately before and after the call, which would result in two code locations with the same source. The listed function becomes the new default function for breakpoint setting (via the breakpoint) command.

list-breakpoints (or lb)
: List all currently active breakpoints with their breakpoint number.

delete-breakpoint (or db) {*number*}
: Delete a breakpoint specified by its breakpoint number. If no number is specified, delete all breakpoints.

step
: Step to the next possible breakpoint location in the current function. This always steps over function calls, instead of stepping into them

3.9.1 Breakpoint Example

Consider this definition of the factorial function:

```
(defun ! (n)
  (if (zerop n)
      1
      (* n (! (1- n)))))
```

This debugger session demonstrates the use of breakpoints:

```
common-lisp-user> (break) ; Invoke debugger

Break

Restarts:
  0: [CONTINUE] Return from BREAK.
  1: [ABORT   ] Return to Top-Level.

Debug  (type H for help)

(INTERACTIVE-EVAL (BREAK))
0] ll #'!
0: #'(LAMBDA (N) (BLOCK ! (IF # 1 #)))
1: (ZEROP N)
2: (* N (! (1- N)))
```

```
3: (1- N)
4: (! (1- N))
5: (* N (! (1- N)))
6: #'(LAMBDA (N) (BLOCK ! (IF # 1 #)))
0] br 2
(* N (! (1- N)))
1: 2 in !
Added.
0] q

common-lisp-user> (! 10) ; Call the function

*Breakpoint hit*

Restarts:
  0: [CONTINUE] Return from BREAK.
  1: [ABORT   ] Return to Top-Level.

Debug  (type H for help)

(! 10) ; We are now in first call (arg 10) before the multiply
Source: (* N (! (1- N)))
3] st

*Step*

(! 10) ; We have finished evaluation of (1- n)
Source: (1- N)
3] st

*Breakpoint hit*

Restarts:
  0: [CONTINUE] Return from BREAK.
  1: [ABORT   ] Return to Top-Level.

Debug  (type H for help)

(! 9) ; We hit the breakpoint in the recursive call
Source: (* N (! (1- N)))
3]
```

3.10 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 or exit.

trace {*option global-value*}* {*name* {*option value*}* }* [*Macro*]

trace is a debugging tool that prints information when specified functions are called. In its simplest form:

(trace *name-1* *name-2* ...)

trace causes a printout on *trace-output* each time that one of the named functions is entered or returns (the *names* are not evaluated.) Trace output is indented according to the number of pending traced calls, and this trace depth is printed at the beginning of each line of output. Printing verbosity of arguments and return values is controlled by *debug-print-level* and *debug-print-length*.

If no *names* or *options* are are given, trace returns the list of all currently traced functions, *traced-function-list*.

Trace options can cause the normal printout to be suppressed, or cause extra information to be printed. Each option is a pair of an option keyword and a value form. Options may be interspersed with function names. Options only affect tracing of the function whose name they appear immediately after. Global options are specified before the first name, and affect all functions traced by a given use of trace. If an already traced function is traced again, any new options replace the old options. The following options are defined:

:condition *form*, :condition-after *form*, :condition-all *form*
: If :condition is specified, then trace does nothing unless *form* evaluates to true at the time of the call. :condition-after is similar, but suppresses the initial printout, and is tested when the function returns. :condition-all tries both before and after.

:wherein *names*
: If specified, *names* is a function name or list of names. trace does nothing unless a call to one of those functions encloses the call to this function (i.e. it would appear in a backtrace.) Anonymous functions have string names like "DEFUN FOO".

:break *form*, :break-after *form*, :break-all *form*
: If specified, and *form* evaluates to true, then the debugger is invoked at the start of the function, at the end of the function, or both, according to the respective option.

:print *form*, :print-after *form*, :print-all *form*
: In addition to the usual printout, the result of evaluating *form* is printed at the start of the function, at the end of the function, or both, according to the respective option. Multiple print options cause multiple values to be printed.

:function *function-form*
: This is a not really an option, but rather another way of specifying what function to trace. The *function-form* is evaluated immediately, and the resulting function is traced.

:encapsulate {*:default* | *t* | *nil*}
: In CMUCL, tracing can be done either by temporarily redefining the function name (encapsulation), or using breakpoints. When breakpoints are used, the function object itself is destructively modified to cause the tracing action. The advantage of using breakpoints is that tracing works even when the function is anonymously called via funcall.

  When :encapsulate is true, tracing is done via encapsulation. :default is the default, and means to use encapsulation for interpreted functions and funcallable instances, breakpoints otherwise. When encapsulation is used, forms are *not* evaluated in the function's lexical environment, but debug:arg can still be used.

:condition, :break and :print forms are evaluated in the lexical environment of the called function; debug:var and debug:arg can be used. The -after and -all forms are evaluated in the null environment.

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

This macro turns off tracing for the specified functions, and removes their names from *traced-function-list*. If no *function-names* are given, then all currently traced functions 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:*max-trace-indentation* [*Variable*]

The maximum number of spaces which should be used to indent trace printout. This variable is initially set to 40.

3.10.1 Encapsulation Functions

The encapsulation functions provide a 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. The Common Lisp fdefinition function always returns the original definition, stripping off any encapsulation.

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.

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.

3.11 Specials

These are the special variables that control the debugger action.

debug:*debug-print-level* [*Variable*]
debug:*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 values of *print-level* and *print-length* are in effect. These variables are initially set to 3 and 5, respectively.

Chapter 4

The Compiler

4.1 Compiler Introduction

This chapter contains information about the compiler that every CMUCL user should be familiar with. Chapter 5 goes into greater depth, describing ways to use more advanced features.

The CMUCL compiler (also known as Python, not to be confused with the programming language of the same name) has many features that are seldom or never supported by conventional Common Lisp compilers:

- Source level debugging of compiled code (see chapter 3.)
- Type error compiler warnings for type errors detectable at compile time.
- Compiler error messages that provide a good indication of where the error appeared in the source.
- Full run-time checking of all potential type errors, with optimization of type checks to minimize the cost.
- Scheme-like features such as proper tail recursion and extensive source-level optimization.
- Advanced tuning and optimization features such as comprehensive efficiency notes, flow analysis, and untagged number representations (see chapter 5.)

4.2 Calling the Compiler

Functions may be compiled using compile, compile-file, or compile-from-stream.

compile *name* &optional *definition* [*Function*]

This function compiles the function whose name is *name*. If *name* is nil, the compiled function object is returned. If *definition* is supplied, it should be a lambda expression that is to be compiled and then placed in the function cell of *name*. As per the proposed X3J13 cleanup "compile-argument-problems", *definition* may also be an interpreted function.

The return values are as per the proposed X3J13 cleanup "compiler-diagnostics". The first value is the function name or function object. The second value is nil if no compiler diagnostics were issued, and t otherwise. The third value is nil if no compiler diagnostics other than style warnings were issued. A non-nil value indicates that there were "serious" compiler diagnostics issued, or that other conditions of type error or warning (but not style-warning) were signaled during compilation.

compile-file *input-pathname* &key :output-file :error-file :trace-file [*Function*]
:error-output :verbose :print :progress
:load :block-compile :entry-points
:byte-compile

The CMUCL compile-file is extended through the addition of several new keywords and an additional interpretation of *input-pathname*:

input-pathname
If this argument is a list of input files, rather than a single input pathname, then all the source files are compiled into a single object file. In this case, the name of the first file is used to determine the default output file names. This is especially useful in combination with *block-compile*.

:output-file
This argument specifies the name of the output file. t gives the default name, nil suppresses the output file.

:error-file
A listing of all the error output is directed to this file. If there are no errors, then no error file is produced (and any existing error file is deleted.) t gives "*name*.err" (the default), and nil suppresses the output file.

:error-output
If t (the default), then error output is sent to *error-output*. If a stream, then output is sent to that stream instead. If nil, then error output is suppressed. Note that this error output is in addition to (but the same as) the output placed in the *error-file*.

:verbose
If t (the default), then the compiler prints to error output at the start and end of compilation of each file. See *compile-verbose* (page 39).

:print
If t (the default), then the compiler prints to error output when each function is compiled. See *compile-print* (page 39).

:progress
If t (default nil), then the compiler prints to error output progress information about the phases of compilation of each function. This is a CMUCL extension that is useful mainly in large block compilations. See *compile-progress* (page 39).

:trace-file
If t, several of the intermediate representations (including annotated assembly code) are dumped out to this file. t gives "*name*.trace". Trace output is off by default. See section 5.12.5, page 92.

:load
If t, load the resulting output file.

:block-compile
Controls the compile-time resolution of function calls. By default, only self-recursive calls are resolved, unless an ext:block-start declaration appears in the source file. See section 5.7.3, page 76.

:entry-points
If non-null, then this is a list of the names of all functions in the file that should have global definitions installed (because they are referenced in other files.) See section 5.7.3, page 76.

:byte-compile
If t, compiling to a compact interpreted byte code is enabled. Possible values are t, nil, and :maybe (the default.) See *byte-compile-default* (page 82) and See section 5.9, page 81.

The return values are as per the proposed X3J13 cleanup "compiler-diagnostics". The first value from compile-file is the truename of the output file, or nil if the file could not be created. The interpretation of the second and third values is described above for compile.

compile-verbose [*Variable*]
compile-print [*Variable*]
compile-progress [*Variable*]

These variables determine the default values for the :verbose, :print and :progress arguments to compile-file.

extensions:compile-from-stream *input-stream* &key :error-stream :trace-stream :block-compile :entry-points :byte-compile [*Function*]

This function is similar to compile-file, but it takes all its arguments as streams. It reads Common Lisp code from *input-stream* until end of file is reached, compiling into the current environment. This function returns the same two values as the last two values of compile. No output files are produced.

4.3 Compilation Units

CMUCL supports the with-compilation-unit macro added to the language by the proposed X3J13 "with-compilation-unit" compiler cleanup. This provides a mechanism for eliminating spurious undefined warnings when there are forward references across files, and also provides a standard way to access compiler extensions.

with-compilation-unit ({*key value*}*) {*form*}* [*Macro*]

This macro evaluates the *forms* in an environment that causes warnings for undefined variables, functions and types to be delayed until all the forms have been evaluated. Each keyword *value* is an evaluated form. These keyword options are recognized:

:override If uses of with-compilation-unit are dynamically nested, the outermost use will take precedence, suppressing printing of undefined warnings by inner uses. However, when the override option is true this shadowing is inhibited; an inner use will print summary warnings for the compilations within the inner scope.

:optimize This is a CMUCL extension that specifies of the "global" compilation policy for the dynamic extent of the body. The argument should evaluate to an optimize declare form, like:

```
(optimize (speed 3) (safety 0))
```

See section 4.7.1, page 50

:optimize-interface
Similar to :optimize, but specifies the compilation policy for function interfaces (argument count and type checking) for the dynamic extent of the body. See section 4.7.2, page 51.

:context-declarations
This is a CMUCL extension that pattern-matches on function names, automatically splicing in any appropriate declarations at the head of the function definition. See section 5.7.5, page 77.

4.3.1 Undefined Warnings

Warnings about undefined variables, functions and types are delayed until the end of the current compilation unit. The compiler entry functions (compile, etc.) implicitly use with-compilation-unit, so undefined warnings will be printed at the end of the compilation unless there is an enclosing with-compilation-unit. In order the gain the benefit of this mechanism, you should wrap a single with-compilation-unit around the calls to compile-file, i.e.:

```
(with-compilation-unit ()
  (compile-file "file1")
  (compile-file "file2")
  ...)
```

Unlike for functions and types, undefined warnings for variables are not suppressed when a definition (e.g. defvar) appears after the reference (but in the same compilation unit.) This is because doing special declarations out of order just doesn't work—although early references will be compiled as special, bindings will be done lexically.

Undefined warnings are printed with full source context (see section 4.4, page 41), which tremendously simplifies the problem of finding undefined references that resulted from macroexpansion. After printing detailed information about the undefined uses of each name, with-compilation-unit also prints summary listings of the names of all the undefined functions, types and variables.

undefined-warning-limit [*Variable*]

This variable controls the number of undefined warnings for each distinct name that are printed with full source context when the compilation unit ends. If there are more undefined references than this, then they are condensed into a single warning:

```
Warning: count more uses of undefined function name.
```

When the value is 0, then the undefined warnings are not broken down by name at all: only the summary listing of undefined names is printed.

4.4 Interpreting Error Messages

One of Python's unique features is the level of source location information it provides in error messages. The error messages contain a lot of detail in a terse format, to they may be confusing at first. Error messages will be illustrated using this example program:

```
(defmacro zoq (x)
  `(roq (ploq (+ ,x 3))))

(defun foo (y)
  (declare (symbol y))
  (zoq y))
```

The main problem with this program is that it is trying to add 3 to a symbol. Note also that the functions roq and ploq aren't defined anywhere.

4.4.1 The Parts of the Error Message

The compiler will produce this warning:

```
File: /usr/me/stuff.lisp
In: DEFUN FOO
  (ZOQ Y)
--> ROQ PLOQ +
==>
  Y
Warning: Result is a SYMBOL, not a NUMBER.
```

In this example we see each of the six possible parts of a compiler error message:

File: /usr/me/stuff.lisp
: This is the *file* that the compiler read the relevant code from. The file name is displayed because it may not be immediately obvious when there is an error during compilation of a large system, especially when with-compilation-unit is used to delay undefined warnings.

In: DEFUN FOO
: This is the *definition* or top-level form responsible for the error. It is obtained by taking the first two elements of the enclosing form whose first element is a symbol beginning with "DEF". If there is no enclosing *def*mumble, then the outermost form is used. If there are multiple *def*mumbles, then they are all printed from the out in, separated by =>'s. In this example, the problem was in the defun for foo.

(ZOQ Y)
: This is the *original source* form responsible for the error. Original source means that the form directly appeared in the original input to the compiler, i.e. in the lambda passed to compile or the top-level form read from the source file. In this example, the expansion of the zoq macro was responsible for the error.

–> ROQ PLOQ +
: This is the *processing path* that the compiler used to produce the errorful code. The processing path is a representation of the evaluated forms enclosing the actual source that the compiler encountered when processing the original source. The path is the first element of each form, or the form itself if the form is not a list. These forms result from the expansion of macros or source-to-source transformation done by the compiler. In this example, the enclosing evaluated forms are the calls to roq, ploq and +. These calls resulted from the expansion of the zoq macro.

==> Y This is the *actual source* responsible for the error. If the actual source appears in the explanation, then we print the next enclosing evaluated form, instead of printing the actual source twice. (This is the form that would otherwise have been the last form of the processing path.) In this example, the problem is with the evaluation of the reference to the variable y.

Warning: Result is a SYMBOL, not a NUMBER.
This is the *explanation* the problem. In this example, the problem is that y evaluates to a symbol, but is in a context where a number is required (the argument to +).

Note that each part of the error message is distinctively marked:

- File: and In: mark the file and definition, respectively.
- The original source is an indented form with no prefix.
- Each line of the processing path is prefixed with ->.
- The actual source form is indented like the original source, but is marked by a preceding ==> line. This is like the "macroexpands to" notation used in *Common Lisp: The Language*.
- The explanation is prefixed with the error severity (see section 4.4.4, page 44), either Error:, Warning:, or Note:.

Each part of the error message is more specific than the preceding one. If consecutive error messages are for nearby locations, then the front part of the error messages would be the same. In this case, the compiler omits as much of the second message as in common with the first. For example:

```
File: /usr/me/stuff.lisp
In: DEFUN FOO
  (ZOQ Y)
--> ROQ
==>
  (PLOQ (+ Y 3))
Warning: Undefined function: PLOQ

==>
  (ROQ (PLOQ (+ Y 3)))
Warning: Undefined function: ROQ
```

In this example, the file, definition and original source are identical for the two messages, so the compiler omits them in the second message. If consecutive messages are entirely identical, then the compiler prints only the first message, followed by:

```
[Last message occurs repeats times]
```

where *repeats* is the number of times the message was given.

If the source was not from a file, then no file line is printed. If the actual source is the same as the original source, then the processing path and actual source will be omitted. If no forms intervene between the original source and the actual source, then the processing path will also be omitted.

4.4.2 The Original and Actual Source

The *original source* displayed will almost always be a list. If the actual source for an error message is a symbol, the original source will be the immediately enclosing evaluated list form. So even if the offending symbol does appear in the original source, the compiler will print the enclosing list and then print the symbol as the actual source (as though the symbol were introduced by a macro.)

When the *actual source* is displayed (and is not a symbol), it will always be code that resulted from the expansion of a macro or a source-to-source compiler optimization. This is code that did not appear in the original source program; it was introduced by the compiler.

Keep in mind that when the compiler displays a source form in an error message, it always displays the most specific (innermost) responsible form. For example, compiling this function:

```
(defun bar (x)
  (let (a)
    (declare (fixnum a))
    (setq a (foo x))
    a))
```

gives this error message:

```
In: DEFUN BAR
  (LET (A) (DECLARE (FIXNUM A)) (SETQ A (FOO X)) A)
Warning: The binding of A is not a FIXNUM:
  NIL
```

This error message is not saying "there's a problem somewhere in this let"—it is saying that there is a problem with the let itself. In this example, the problem is that a's nil initial value is not a fixnum.

4.4.3 The Processing Path

The processing path is mainly useful for debugging macros, so if you don't write macros, you can ignore the processing path. Consider this example:

```
(defun foo (n)
  (dotimes (i n *undefined*)))
```

Compiling results in this error message:

```
In: DEFUN FOO
  (DOTIMES (I N *UNDEFINED*))
--> DO BLOCK LET TAGBODY RETURN-FROM
==>
  (PROGN *UNDEFINED*)
Warning: Undefined variable: *UNDEFINED*
```

Note that do appears in the processing path. This is because dotimes expands into:

```
(do ((i 0 (1+ i)) (#:g1 n))
    ((>= i #:g1) *undefined*)
  (declare (type unsigned-byte i)))
```

The rest of the processing path results from the expansion of do:

```
(block nil
  (let ((i 0) (#:g1 n))
    (declare (type unsigned-byte i))
    (tagbody (go #:g3)
     #:g2    (psetq i (1+ i))
     #:g3    (unless (>= i #:g1) (go #:g2))
             (return-from nil (progn *undefined*)))))
```

In this example, the compiler descended into the block, let, tagbody and return-from to reach the progn printed as the actual source. This is a place where the "actual source appears in explanation" rule was applied. The innermost actual source form was the symbol *undefined* itself, but that also appeared in the explanation, so the compiler backed out one level.

4.4.4 Error Severity

There are three levels of compiler error severity:

Error — This severity is used when the compiler encounters a problem serious enough to prevent normal processing of a form. Instead of compiling the form, the compiler compiles a call to error. Errors are used mainly for signaling syntax errors. If an error happens during macroexpansion, the compiler will handle it. The compiler also handles and attempts to proceed from read errors.

Warning — Warnings are used when the compiler can prove that something bad will happen if a portion of the program is executed, but the compiler can proceed by compiling code that signals an error at runtime if the problem has not been fixed:

- Violation of type declarations, or
- Function calls that have the wrong number of arguments or malformed keyword argument lists, or
- Referencing a variable declared ignore, or unrecognized declaration specifiers.

In the language of the Common Lisp standard, these are situations where the compiler can determine that a situation with undefined consequences or that would cause an error to be signaled would result at runtime.

Note — Notes are used when there is something that seems a bit odd, but that might reasonably appear in correct programs.

Note that the compiler does not fully conform to the proposed X3J13 "compiler-diagnostics" cleanup. Errors, warnings and notes mostly correspond to errors, warnings and style-warnings, but many things that the cleanup considers to be style-warnings are printed as warnings rather than notes. Also, warnings, style-warnings and most errors aren't really signaled using the condition system.

4.4.5 Errors During Macroexpansion

The compiler handles errors that happen during macroexpansion, turning them into compiler errors. If you want to debug the error (to debug a macro), you can set *break-on-signals* to error. For example, this definition:

```
(defun foo (e l)
  (do ((current l (cdr current))
       ((atom current) nil))
      (when (eq (car current) e) (return current))))
```

gives this error:

```
In: DEFUN FOO
  (DO ((CURRENT L #) (# NIL)) (WHEN (EQ # E) (RETURN CURRENT)) )
Error: (during macroexpansion)

Error in function LISP::DO-DO-BODY.
DO step variable is not a symbol: (ATOM CURRENT)
```

4.4.6 Read Errors

The compiler also handles errors while reading the source. For example:

```
Error: Read error at 2:
 "(,/\foo)"
Error in function LISP::COMMA-MACRO.
Comma not inside a backquote.
```

The "at 2" refers to the character position in the source file at which the error was signaled, which is generally immediately after the erroneous text. The next line, "(,/\foo)", is the line in the source that contains the error file position. The "/\ " indicates the error position within that line (in this example, immediately after the offending comma.)

When in Hemlock (or any other EMACS-like editor), you can go to a character position with:

```
M-< C-u position C-f
```

Note that if the source is from a Hemlock buffer, then the position is relative to the start of the compiled region or defun, not the file or buffer start.

After printing a read error message, the compiler attempts to recover from the error by backing up to the start of the enclosing top-level form and reading again with *read-suppress* true. If the compiler can recover from the error, then it substitutes a call to cerror for the unreadable form and proceeds to compile the rest of the file normally.

If there is a read error when the file position is at the end of the file (i.e., an unexpected EOF error), then the error message looks like this:

```
Error: Read error in form starting at 14:
 "(defun test ()"
Error in function LISP::FLUSH-WHITESPACE.
EOF while reading #<Stream for file "/usr/me/test.lisp">
```

In this case, "starting at 14" indicates the character position at which the compiler started reading, i.e. the position before the start of the form that was missing the closing delimiter. The line "(defun test ()" is first line after the starting position that the compiler thinks might contain the unmatched open delimiter.

4.4.7 Error Message Parameterization

There is some control over the verbosity of error messages. See also *undefined-warning-limit* (page 40), *efficiency-note-limit* and *efficiency-note-cost-threshold* (page 95).

enclosing-source-cutoff [*Variable*]

This variable specifies the number of enclosing actual source forms that are printed in full, rather than in the abbreviated processing path format. Increasing the value from its default of 1 allows you to see more of the guts of the macroexpanded source, which is useful when debugging macros.

error-print-length [*Variable*]
error-print-level [*Variable*]

These variables are the print level and print length used in printing error messages. The default values are 5 and 3. If null, the global values of *print-level* and *print-length* are used.

extensions:def-source-context *name lambda-list* {*form*}* [*Macro*]

This macro defines how to extract an abbreviated source context from the *name*d form when it appears in the compiler input. *lambda-list* is a defmacro style lambda-list used to parse the arguments. The *body* should return a list of subforms that can be printed on about one line. There are predefined methods for defstruct, defmethod, etc. If no method is defined, then the first two subforms are returned. Note that this facility implicitly determines the string name associated with anonymous functions.

4.5 Types in Python

A big difference between Python and all other Common Lisp compilers is the approach to type checking and amount of knowledge about types:

- Python treats type declarations much differently that other Lisp compilers do. Python doesn't blindly believe type declarations; it considers them assertions about the program that should be checked.
- Python also has a tremendously greater knowledge of the Common Lisp type system than other compilers. Support is incomplete only for the not, and and satisfies types.

See also sections 5.2 and 5.3.

4.5.1 Compile Time Type Errors

If the compiler can prove at compile time that some portion of the program cannot be executed without a type error, then it will give a warning at compile time. It is possible that the offending code would never actually be executed at run-time due to some higher level consistency constraint unknown to the compiler, so a type warning doesn't always indicate an incorrect program. For example, consider this code fragment:

```
(defun raz (foo)
  (let ((x (case foo
             (:this 13)
             (:that 9)
             (:the-other 42))))
    (declare (fixnum x))
    (foo x)))
```

Compilation produces this warning:

```
In: DEFUN RAZ
  (CASE FOO (:THIS 13) (:THAT 9) (:THE-OTHER 42))
--> LET COND IF COND IF COND IF
==>
  (COND)
Warning: This is not a FIXNUM:
  NIL
```

In this case, the warning is telling you that if foo isn't any of :this, :that or :the-other, then x will be initialized to nil, which the fixnum declaration makes illegal. The warning will go away if ecase is used instead of case, or if :the-other is changed to t.

This sort of spurious type warning happens moderately often in the expansion of complex macros and in inline functions. In such cases, there may be dead code that is impossible to correctly execute. The compiler can't always prove this code is dead (could never be executed), so it compiles the erroneous code (which will always signal an error if it is executed) and gives a warning.

extensions:required-argument [*Function*]

This function can be used as the default value for keyword arguments that must always be supplied. Since it is known by the compiler to never return, it will avoid any compile-time type warnings that would result from a default value inconsistent with the declared type. When this function is called, it signals an error indicating that a required keyword argument was not supplied. This function is also useful for defstruct slot defaults corresponding to required arguments. See section 5.2.5, page 57.

Although this function is a CMUCL extension, it is relatively harmless to use it in otherwise portable code, since you can easily define it yourself:

```
(defun required-argument ()
  (error "A required keyword argument was not supplied."))
```

Type warnings are inhibited when the extensions:inhibit-warnings optimization quality is 3 (see section 4.7, page 50.) This can be used in a local declaration to inhibit type warnings in a code fragment that has spurious warnings.

4.5.2 Precise Type Checking

With the default compilation policy, all type assertions[1] are precisely checked. Precise checking means that the check is done as though typep had been called with the exact type specifier that appeared in the declaration. Python uses *policy* to determine whether to trust type assertions (see section 4.7, page 50). Type assertions from declarations are indistinguishable from the type assertions on arguments to built-in functions. In Python, adding type declarations makes code safer.

If a variable is declared to be (integer 3 17), then its value must always always be an integer between 3 and 17. If multiple type declarations apply to a single variable, then all the declarations must be correct; it is as though all the types were intersected producing a single and type specifier.

Argument type declarations are automatically enforced. If you declare the type of a function argument, a type check will be done when that function is called. In a function call, the called function does the argument type checking, which means that a more restrictive type assertion in the calling function (e.g., from the) may be lost.

The types of structure slots are also checked. The value of a structure slot must always be of the type indicated in any :type slot option.[2] Because of precise type checking, the arguments to slot accessors are checked to be the correct type of structure.

In traditional Common Lisp compilers, not all type assertions are checked, and type checks are not precise. Traditional compilers blindly trust explicit type declarations, but may check the argument type assertions for built-in functions. Type checking is not precise, since the argument type checks will be for the most general type legal for that argument. In many systems, type declarations suppress what little type checking is being done, so adding type declarations makes code unsafe. This is a problem since it discourages writing type declarations during initial coding. In addition to being more error prone, adding type declarations during tuning also loses all the benefits of debugging with checked type assertions.

To gain maximum benefit from Python's type checking, you should always declare the types of function arguments and structure slots as precisely as possible. This often involves the use of or, member and other list-style type specifiers. Paradoxically, even though adding type declarations introduces type checks, it usually reduces the overall amount of type checking. This is especially true for structure slot type declarations.

Python uses the safety optimization quality (rather than presence or absence of declarations) to choose one of three levels of run-time type error checking: see section 4.7.1, page 50. See section 5.2, page 56 for more information about types in Python.

4.5.3 Weakened Type Checking

When the value for the speed optimization quality is greater than safety, and safety is not 0, then type checking is weakened to reduce the speed and space penalty. In structure-intensive code this can double the speed, yet still catch most type errors. Weakened type checks provide a level of safety similar to that of "safe" code in other Common Lisp compilers.

A type check is weakened by changing the check to be for some convenient supertype of the asserted type. For example, (integer 3 17) is changed to fixnum, (simple-vector 17) to simple-vector, and structure types are changed to structure. A complex check like:

```
(or node hunk (member :foo :bar :baz))
```

will be omitted entirely (i.e., the check is weakened to *.) If a precise check can be done for no extra cost, then no weakening is done.

Although weakened type checking is similar to type checking done by other compilers, it is sometimes safer and sometimes less safe. Weakened checks are done in the same places is precise checks, so all the preceding

[1] There are a few circumstances where a type declaration is discarded rather than being used as type assertion. This doesn't affect safety much, since such discarded declarations are also not believed to be true by the compiler.

[2] The initial value need not be of this type as long as the corresponding argument to the constructor is always supplied, but this will cause a compile-time type warning unless required-argument is used.

discussion about where checking is done still applies. Weakened checking is sometimes somewhat unsafe because although the check is weakened, the precise type is still input into type inference. In some contexts this will result in type inferences not justified by the weakened check, and hence deletion of some type checks that would be done by conventional compilers.

For example, if this code was compiled with weakened checks:

```
(defstruct foo
  (a nil :type simple-string))

(defstruct bar
  (a nil :type single-float))

(defun myfun (x)
  (declare (type bar x))
  (* (bar-a x) 3.0))
```

and myfun was passed a foo, then no type error would be signaled, and we would try to multiply a simple-vector as though it were a float (with unpredictable results.) This is because the check for bar was weakened to structure, yet when compiling the call to bar-a, the compiler thinks it knows it has a bar.

Note that normally even weakened type checks report the precise type in error messages. For example, if myfun's bar check is weakened to structure, and the argument is nil, then the error will be:

```
Type-error in MYFUN:
  NIL is not of type BAR
```

However, there is some speed and space cost for signaling a precise error, so the weakened type is reported if the speed optimization quality is 3 or debug quality is less than 1:

```
Type-error in MYFUN:
  NIL is not of type STRUCTURE
```

See section 4.7.1, page 50 for further discussion of the optimize declaration.

4.6 Getting Existing Programs to Run

Since Python does much more comprehensive type checking than other Lisp compilers, Python will detect type errors in many programs that have been debugged using other compilers. These errors are mostly incorrect declarations, although compile-time type errors can find actual bugs if parts of the program have never been tested.

Some incorrect declarations can only be detected by run-time type checking. It is very important to initially compile programs with full type checks and then test this version. After the checking version has been tested, then you can consider weakening or eliminating type checks. **This applies even to previously debugged programs.** Python does much more type inference than other Common Lisp compilers, so believing an incorrect declaration does much more damage.

The most common problem is with variables whose initial value doesn't match the type declaration. Incorrect initial values will always be flagged by a compile-time type error, and they are simple to fix once located. Consider this code fragment:

```
(prog (foo)
  (declare (fixnum foo))
  (setq foo ...)
  ...)
```

Here the variable foo is given an initial value of nil, but is declared to be a fixnum. Even if it is never read, the initial value of a variable must match the declared type. There are two ways to fix this problem. Change the declaration:

```
(prog (foo)
  (declare (type (or fixnum null) foo))
  (setq foo ...)
  ...)
```

or change the initial value:

```
(prog ((foo 0))
  (declare (fixnum foo))
  (setq foo ...)
  ...)
```

It is generally preferable to change to a legal initial value rather than to weaken the declaration, but sometimes it is simpler to weaken the declaration than to try to make an initial value of the appropriate type.

Another declaration problem occasionally encountered is incorrect declarations on defmacro arguments. This probably usually happens when a function is converted into a macro. Consider this macro:

```
(defmacro my-1+ (x)
  (declare (fixnum x))
  `(the fixnum (1+ ,x)))
```

Although legal and well-defined Common Lisp, this meaning of this definition is almost certainly not what the writer intended. For example, this call is illegal:

```
(my-1+ (+ 4 5))
```

The call is illegal because the argument to the macro is (+ 4 5), which is a list, not a fixnum. Because of macro semantics, it is hardly ever useful to declare the types of macro arguments. If you really want to assert something about the type of the result of evaluating a macro argument, then put a the in the expansion:

```
(defmacro my-1+ (x)
  `(the fixnum (1+ (the fixnum ,x))))
```

In this case, it would be stylistically preferable to change this macro back to a function and declare it inline. Macros have no efficiency advantage over inline functions when using Python. See section 5.8, page 79.

Some more subtle problems are caused by incorrect declarations that can't be detected at compile time. Consider this code:

```
(do ((pos 0 (position #\a string :start (1+ pos))))
    ((null pos))
  (declare (fixnum pos))
  ...)
```

Although pos is almost always a fixnum, it is nil at the end of the loop. If this example is compiled with full type checks (the default), then running it will signal a type error at the end of the loop. If compiled without type checks, the program will go into an infinite loop (or perhaps position will complain because (1+ nil) isn't a sensible start.) Why? Because if you compile without type checks, the compiler just quietly believes the type declaration. Since pos is always a fixnum, it is never nil, so (null pos) is never true, and the loop exit test is optimized away. Such errors are sometimes flagged by unreachable code notes (see section 5.4.5, page 68), but it is still important to initially compile any system with full type checks, even if the system works fine when compiled using other compilers.

In this case, the fix is to weaken the type declaration to (or fixnum null).[3] Note that there is usually little performance penalty for weakening a declaration in this way. Any numeric operations in the body can still assume the variable is a fixnum, since nil is not a legal numeric argument. Another possible fix would be to say:

[3] Actually, this declaration is totally unnecessary in Python, since it already knows position returns a non-negative fixnum or nil.

```
(do ((pos 0 (position #\a string :start (1+ pos))))
    ((null pos))
  (let ((pos pos))
    (declare (fixnum pos))
    ...))
```

This would be preferable in some circumstances, since it would allow a non-standard representation to be used for the local pos variable in the loop body (see section 5.11.3.)

In summary, remember that *all* values that a variable *ever* has must be of the declared type, and that you should test using safe code initially.

4.7 Compiler Policy

The policy is what tells the compiler *how* to compile a program. This is logically (and often textually) distinct from the program itself. Broad control of policy is provided by the optimize declaration; other declarations and variables control more specific aspects of compilation.

4.7.1 The Optimize Declaration

The optimize declaration recognizes six different *qualities*. The qualities are conceptually independent aspects of program performance. In reality, increasing one quality tends to have adverse effects on other qualities. The compiler compares the relative values of qualities when it needs to make a trade-off; i.e., if speed is greater than safety, then improve speed at the cost of safety.

The default for all qualities (except debug) is 1. Whenever qualities are equal, ties are broken according to a broad idea of what a good default environment is supposed to be. Generally this downplays speed, compile-speed and space in favor of safety and debug. Novice and casual users should stick to the default policy. Advanced users often want to improve speed and memory usage at the cost of safety and debuggability.

If the value for a quality is 0 or 3, then it may have a special interpretation. A value of 0 means "totally unimportant", and a 3 means "ultimately important." These extreme optimization values enable "heroic" compilation strategies that are not always desirable and sometimes self-defeating. Specifying more than one quality as 3 is not desirable, since it doesn't tell the compiler which quality is most important.

These are the optimization qualities:

speed
: How fast the program should is run. speed 3 enables some optimizations that hurt debuggability.

compilation-speed
: How fast the compiler should run. Note that increasing this above safety weakens type checking.

space
: How much space the compiled code should take up. Inline expansion is mostly inhibited when space is greater than speed. A value of 0 enables promiscuous inline expansion. Wide use of a 0 value is not recommended, as it may waste so much space that run time is slowed. See section 5.8, page 79 for a discussion of inline expansion.

debug
: How debuggable the program should be. The quality is treated differently from the other qualities: each value indicates a particular level of debugger information; it is not compared with the other qualities. See section 3.6, page 32 for more details.

safety
: How much error checking should be done. If speed, space or compilation-speed is more important than safety, then type checking is weakened (see section 4.5.3, page 47). If safety if 0, then no run time error checking is done. In addition to suppressing type checks, 0 also suppresses argument count checking, unbound-symbol checking and array bounds checks.

extensions:inhibit-warnings
: This is a CMUCL extension that determines how little (or how much) diagnostic output should be printed during compilation. This quality is compared to other qualities to determine whether to print style notes and warnings concerning those qualities. If speed is greater than inhibit-warnings, then notes about how to improve speed will be printed, etc. The default value is 1, so raising the

value for any standard quality above its default enables notes for that quality. If inhibit-warnings is 3, then all notes and most non-serious warnings are inhibited. This is useful with declare to suppress warnings about unavoidable problems.

4.7.2 The Optimize-Interface Declaration

The extensions:optimize-interface declaration is identical in syntax to the optimize declaration, but it specifies the policy used during compilation of code the compiler automatically generates to check the number and type of arguments supplied to a function. It is useful to specify this policy separately, since even thoroughly debugged functions are vulnerable to being passed the wrong arguments. The optimize-interface declaration can specify that arguments should be checked even when the general optimize policy is unsafe.

Note that this argument checking is the checking of user-supplied arguments to any functions defined within the scope of the declaration, not the checking of arguments to Common Lisp primitives that appear in those definitions.

The idea behind this declaration is that it allows the definition of functions that appear fully safe to other callers, but that do no internal error checking. Of course, it is possible that arguments may be invalid in ways other than having incorrect type. Functions compiled unsafely must still protect themselves against things like user-supplied array indices that are out of bounds and improper lists. See also the :context-declarations option to with-compilation-unit (page 40).

4.8 Open Coding and Inline Expansion

Since Common Lisp forbids the redefinition of standard functions[4], the compiler can have special knowledge of these standard functions embedded in it. This special knowledge is used in various ways (open coding, inline expansion, source transformation), but the implications to the user are basically the same:

- Attempts to redefine standard functions may be frustrated, since the function may never be called. Although it is technically illegal to redefine standard functions, users sometimes want to implicitly redefine these functions when they are debugging using the trace macro. Special-casing of standard functions can be inhibited using the notinline declaration.

- The compiler can have multiple alternate implementations of standard functions that implement different trade-offs of speed, space and safety. This selection is based on the compiler policy, see section 4.7, page 50.

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, then

```
(nth x l)
```

might stay the same, or turn into something like:

[4] See the proposed X3J13 “lisp-symbol-redefinition” cleanup.

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

In general, open coding sacrifices space for speed, but some functions (such as car) are so simple that they are always open-coded. Even when not open-coded, a call to a standard function may be transformed into a different function call (as in the last example) or compiled as *static call*. Static function call uses a more efficient calling convention that forbids redefinition.

Chapter 5

Advanced Compiler Use and Efficiency Hints

by Robert MacLachlan

5.1 Advanced Compiler Introduction

In CMUCL, as with any language on any computer, the path to efficient code starts with good algorithms and sensible programming techniques, but to avoid inefficiency pitfalls, you need to know some of this implementation's quirks and features. This chapter is mostly a fairly long and detailed overview of what optimizations Python does. Although there are the usual negative suggestions of inefficient features to avoid, the main emphasis is on describing the things that programmers can count on being efficient.

The optimizations described here can have the effect of speeding up existing programs written in conventional styles, but the potential for new programming styles that are clearer and less error-prone is at least as significant. For this reason, several sections end with a discussion of the implications of these optimizations for programming style.

5.1.1 Types

Python's support for types is unusual in three major ways:

- Precise type checking encourages the specific use of type declarations as a form of run-time consistency checking. This speeds development by localizing type errors and giving more meaningful error messages. See section 4.5.2, page 47. Python produces completely safe code; optimized type checking maintains reasonable efficiency on conventional hardware (see section 5.3.6, page 64.)
- Comprehensive support for the Common Lisp type system makes complex type specifiers useful. Using type specifiers such as `or` and `member` has both efficiency and robustness advantages. See section 5.2, page 56.
- Type inference eliminates the need for some declarations, and also aids compile-time detection of type errors. Given detailed type declarations, type inference can often eliminate type checks and enable more efficient object representations and code sequences. Checking all types results in fewer type checks. See sections 5.3 and 5.11.2.

5.1.2 Optimization

The main barrier to efficient Lisp programs is not that there is no efficient way to code the program in Lisp, but that it is difficult to arrive at that efficient coding. Common Lisp is a highly complex language, and usually has many semantically equivalent "reasonable" ways to code a given problem. It is desirable to make all of these equivalent solutions have comparable efficiency so that programmers don't have to waste time discovering the most efficient solution.

Source level optimization increases the number of efficient ways to solve a problem. This effect is much larger than the increase in the efficiency of the "best" solution. Source level optimization transforms the original program into a more efficient (but equivalent) program. Although the optimizer isn't doing anything the programmer couldn't have done, this high-level optimization is important because:

- The programmer can code simply and directly, rather than obfuscating code to please the compiler.
- When presented with a choice of similar coding alternatives, the programmer can chose whichever happens to be most convenient, instead of worrying about which is most efficient.

Source level optimization eliminates the need for macros to optimize their expansion, and also increases the effectiveness of inline expansion. See sections 5.4 and 5.8.

Efficient support for a safer programming style is the biggest advantage of source level optimization. Existing tuned programs typically won't benefit much from source optimization, since their source has already been optimized by hand. However, even tuned programs tend to run faster under Python because:

- Low level optimization and register allocation provides modest speedups in any program.
- Block compilation and inline expansion can reduce function call overhead, but may require some program restructuring. See sections 5.8, 5.6 and 5.7.
- Efficiency notes will point out important type declarations that are often missed even in highly tuned programs. See section 5.13, page 92.
- Existing programs can be compiled safely without prohibitive speed penalty, although they would be faster and safer with added declarations. See section 5.3.6, page 64.
- The context declaration mechanism allows both space and runtime of large systems to be reduced without sacrificing robustness by semi-automatically varying compilation policy without addition any `optimize` declarations to the source. See section 5.7.5, page 77.
- Byte compilation can be used to dramatically reduce the size of code that is not speed-critical. See section 5.9, page 81

5.1.3 Function Call

The sort of symbolic programs generally written in Common Lisp often favor recursion over iteration, or have inner loops so complex that they involve multiple function calls. Such programs spend a larger fraction of their time doing function calls than is the norm in other languages; for this reason Common Lisp implementations strive to make the general (or full) function call as inexpensive as possible. Python goes beyond this by providing two good alternatives to full call:

- Local call resolves function references at compile time, allowing better calling sequences and optimization across function calls. See section 5.6, page 72.
- Inline expansion totally eliminates call overhead and allows many context dependent optimizations. This provides a safe and efficient implementation of operations with function semantics, eliminating the need for error-prone macro definitions or manual case analysis. Although most Common Lisp implementations support inline expansion, it becomes a more powerful tool with Python's source level optimization. See sections 5.4 and 5.8.

Generally, Python provides simple implementations for simple uses of function call, rather than having only a single calling convention. These features allow a more natural programming style:

- Proper tail recursion. See section 5.5, page 71
- Relatively efficient closures.

- A funcall that is as efficient as normal named call.
- Calls to local functions such as from labels are optimized:
  - Control transfer is a direct jump.
  - The closure environment is passed in registers rather than heap allocated.
  - Keyword arguments and multiple values are implemented more efficiently.

  See section 5.6, page 72.

5.1.4 Representation of Objects

Sometimes traditional Common Lisp implementation techniques compare so poorly to the techniques used in other languages that Common Lisp can become an impractical language choice. Terrible inefficiencies appear in number-crunching programs, since Common Lisp numeric operations often involve number-consing and generic arithmetic. Python supports efficient natural representations for numbers (and some other types), and allows these efficient representations to be used in more contexts. Python also provides good efficiency notes that warn when a crucial declaration is missing.

See section 5.11.2 for more about object representations and numeric types. Also see section 5.13, page 92 about efficiency notes.

5.1.5 Writing Efficient Code

Writing efficient code that works is a complex and prolonged process. It is important not to get so involved in the pursuit of efficiency that you lose sight of what the original problem demands. Remember that:

- The program should be correct—it doesn't matter how quickly you get the wrong answer.
- Both the programmer and the user will make errors, so the program must be robust—it must detect errors in a way that allows easy correction.
- A small portion of the program will consume most of the resources, with the bulk of the code being virtually irrelevant to efficiency considerations. Even experienced programmers familiar with the problem area cannot reliably predict where these "hot spots" will be.

The best way to get efficient code that is still worth using, is to separate coding from tuning. During coding, you should:

- Use a coding style that aids correctness and robustness without being incompatible with efficiency.
- Choose appropriate data structures that allow efficient algorithms and object representations (see section 5.10, page 82). Try to make interfaces abstract enough so that you can change to a different representation if profiling reveals a need.
- Whenever you make an assumption about a function argument or global data structure, add consistency assertions, either with type declarations or explicit uses of assert, ecase, etc.

During tuning, you should:

- Identify the hot spots in the program through profiling (section 5.14.)
- Identify inefficient constructs in the hot spot with efficiency notes, more profiling, or manual inspection of the source. See sections 5.12 and 5.13.
- Add declarations and consider the application of optimizations. See sections 5.6, 5.8 and 5.11.2.
- If all else fails, consider algorithm or data structure changes. If you did a good job coding, changes will be easy to introduce.

5.2 More About Types in Python

This section goes into more detail describing what types and declarations are recognized by Python. The area where Python differs most radically from previous Common Lisp compilers is in its support for types:

- Precise type checking helps to find bugs at run time.
- Compile-time type checking helps to find bugs at compile time.
- Type inference minimizes the need for generic operations, and also increases the efficiency of run time type checking and the effectiveness of compile time type checking.
- Support for detailed types provides a wealth of opportunity for operation-specific type inference and optimization.

5.2.1 More Types Meaningful

Common Lisp has a very powerful type system, but conventional Common Lisp implementations typically only recognize the small set of types special in that implementation. In these systems, there is an unfortunate paradox: a declaration for a relatively general type like fixnum will be recognized by the compiler, but a highly specific declaration such as (integer 3 17) is totally ignored.

This is obviously a problem, since the user has to know how to specify the type of an object in the way the compiler wants it. A very minimal (but rarely satisfied) criterion for type system support is that it be no worse to make a specific declaration than to make a general one. Python goes beyond this by exploiting a number of advantages obtained from detailed type information.

Using more restrictive types in declarations allows the compiler to do better type inference and more compile-time type checking. Also, when type declarations are considered to be consistency assertions that should be verified (conditional on policy), then complex types are useful for making more detailed assertions.

Python "understands" the list-style or, member, function, array and number type specifiers. Understanding means that:

- If the type contains more information than is used in a particular context, then the extra information is simply ignored, rather than derailing type inference.
- In many contexts, the extra information from these type specifier is used to good effect. In particular, type checking in Python is *precise*, so these complex types can be used in declarations to make interesting assertions about functions and data structures (see section 4.5.2, page 47.) More specific declarations also aid type inference and reduce the cost for type checking.

For related information, see section 5.11, page 84 for numeric types, and section 5.10.3 for array types.

5.2.2 Canonicalization

When given a type specifier, Python will often rewrite it into a different (but equivalent) type. This is the mechanism that Python uses for detecting type equivalence. For example, in Python's canonical representation, these types are equivalent:

```
(or list (member :end)) ≡ (or cons (member nil :end))
```

This has two implications for the user:

- The standard symbol type specifiers for atom, null, fixnum, etc., are in no way magical. The null type is actually defined to be (member nil), list is (or cons null), and fixnum is (signed-byte 30).
- When the compiler prints out a type, it may not look like the type specifier that originally appeared in the program. This is generally not a problem, but it must be taken into consideration when reading compiler error messages.

5.2.3 Member Types

The member type specifier can be used to represent "symbolic" values, analogous to the enumerated types of Pascal. For example, the second value of find-symbol has this type:

```
(member :internal :external :inherited nil)
```

Member types are very useful for expressing consistency constraints on data structures, for example:

```
(defstruct ice-cream
  (flavor :vanilla :type (member :vanilla :chocolate :strawberry)))
```

Member types are also useful in type inference, as the number of members can sometimes be pared down to one, in which case the value is a known constant.

5.2.4 Union Types

The or (union) type specifier is understood, and is meaningfully applied in many contexts. The use of or allows assertions to be made about types in dynamically typed programs. For example:

```
(defstruct box
  (next nil :type (or box null))
  (top :removed :type (or box-top (member :removed))))
```

The type assertion on the top slot ensures that an error will be signaled when there is an attempt to store an illegal value (such as :rmoved.) Although somewhat weak, these union type assertions provide a useful input into type inference, allowing the cost of type checking to be reduced. For example, this loop is safely compiled with no type checks:

```
(defun find-box-with-top (box)
  (declare (type (or box null) box))
  (do ((current box (box-next current)))
      ((null current))
    (unless (eq (box-top current) :removed)
      (return current))))
```

Union types are also useful in type inference for representing types that are partially constrained. For example, the result of this expression:

```
(if foo
    (logior x y)
    (list x y))
```

can be expressed as (or integer cons).

5.2.5 The Empty Type

The type nil is also called the empty type, since no object is of type nil. The union of no types, (or), is also empty. Python's interpretation of an expression whose type is nil is that the expression never yields any value, but rather fails to terminate, or is thrown out of. For example, the type of a call to error or a use of return is nil. When the type of an expression is empty, compile-time type warnings about its value are suppressed; presumably somebody else is signaling an error. If a function is declared to have return type nil, but does in fact return, then (in safe compilation policies) a "NIL Function returned" error will be signaled. See also the function required-argument (page 46).

5.2.6 Function Types

function types are understood in the restrictive sense, specifying:

- The argument syntax that the function must be called with. This is information about what argument counts are acceptable, and which keyword arguments are recognized. In Python, warnings about argument syntax are a consequence of function type checking.
- The types of the argument values that the caller must pass. If the compiler can prove that some argument to a call is of a type disallowed by the called function's type, then it will give a compile-time type warning. In addition to being used for compile-time type checking, these type assertions are also used as output type assertions in code generation. For example, if foo is declared to have a fixnum argument, then the 1+ in (foo (1+ x)) is compiled with knowledge that the result must be a fixnum.
- The types the values that will be bound to argument variables in the function's definition. Declaring a function's type with ftype implicitly declares the types of the arguments in the definition. Python checks for consistency between the definition and the ftype declaration. Because of precise type checking, an error will be signaled when a function is called with an argument of the wrong type.
- The type of return value(s) that the caller can expect. This information is a useful input to type inference. For example, if a function is declared to return a fixnum, then when a call to that function appears in an expression, the expression will be compiled with knowledge that the call will return a fixnum.
- The type of return value(s) that the definition must return. The result type in an ftype declaration is treated like an implicit the wrapped around the body of the definition. If the definition returns a value of the wrong type, an error will be signaled. If the compiler can prove that the function returns the wrong type, then it will give a compile-time warning.

This is consistent with the new interpretation of function types and the ftype declaration in the proposed X3J13 "function-type-argument-type-semantics" cleanup. Note also, that if you don't explicitly declare the type of a function using a global ftype declaration, then Python will compute a function type from the definition, providing a degree of inter-routine type inference, see section 5.3.3, page 62.

5.2.7 The Values Declaration

CMUCL supports the values declaration as an extension to Common Lisp. The syntax is (values *type1 type2*...*typen*). This declaration is semantically equivalent to a the form wrapped around the body of the special form in which the values declaration appears. The advantage of values over the is purely syntactic—it doesn't introduce more indentation. For example:

```
(defun foo (x)
  (declare (values single-float))
  (ecase x
    (:this ...)
    (:that ...)
    (:the-other ...)))
```

is equivalent to:

```
(defun foo (x)
  (the single-float
       (ecase x
         (:this ...)
         (:that ...)
         (:the-other ...))))
```

and

```
(defun floor (number &optional (divisor 1))
  (declare (values integer real))
  ...)
```

is equivalent to:

```
(defun floor (number &optional (divisor 1))
  (the (values integer real)
       ...))
```

In addition to being recognized by lambda (and hence by defun), the values declaration is recognized by all the other special forms with bodies and declarations: let, let*, labels and flet. Macros with declarations usually splice the declarations into one of the above forms, so they will accept this declaration too, but the exact effect of a values declaration will depend on the macro.

If you declare the types of all arguments to a function, and also declare the return value types with values, you have described the type of the function. Python will use this argument and result type information to derive a function type that will then be applied to calls of the function (see section 5.2.6, page 58.) This provides a way to declare the types of functions that is much less syntactically awkward than using the ftype declaration with a function type specifier.

Although the values declaration is non-standard, it is relatively harmless to use it in otherwise portable code, since any warning in non-CMU implementations can be suppressed with the standard declaration proclamation.

5.2.8 Structure Types

Because of precise type checking, structure types are much better supported by Python than by conventional compilers:

- The structure argument to structure accessors is precisely checked—if you call foo-a on a bar, an error will be signaled.
- The types of slot values are precisely checked—if you pass the wrong type argument to a constructor or a slot setter, then an error will be signaled.

This error checking is tremendously useful for detecting bugs in programs that manipulate complex data structures.

An additional advantage of checking structure types and enforcing slot types is that the compiler can safely believe slot type declarations. Python effectively moves the type checking from the slot access to the slot setter or constructor call. This is more efficient since caller of the setter or constructor often knows the type of the value, entirely eliminating the need to check the value's type. Consider this example:

```
(defstruct coordinate
  (x nil :type single-float)
  (y nil :type single-float))

(defun make-it ()
  (make-coordinate :x 1.0 :y 1.0))

(defun use-it (it)
  (declare (type coordinate it))
  (sqrt (expt (coordinate-x it) 2) (expt (coordinate-y it) 2)))
```

make-it and use-it are compiled with no checking on the types of the float slots, yet use-it can use single-float arithmetic with perfect safety. Note that make-coordinate must still check the values of x and y unless the call is block compiled or inline expanded (see section 5.6, page 72.) But even without this advantage, it is almost always more efficient to check slot values on structure initialization, since slots are usually written once and read many times.

5.2.9 The Freeze-Type Declaration

The extensions:freeze-type declaration is a CMUCL extension that enables more efficient compilation of user-defined types by asserting that the definition is not going to change. This declaration may only be used globally (with declaim or proclaim). Currently freeze-type only affects structure type testing done by typep, typecase, etc. Here is an example:

```
(declaim (freeze-type foo bar))
```

This asserts that the types foo and bar and their subtypes are not going to change. This allows more efficient type testing, since the compiler can open-code a test for all possible subtypes, rather than having to examine the type hierarchy at run-time.

5.2.10 Type Restrictions

Avoid use of the and, not and satisfies types in declarations, since type inference has problems with them. When these types do appear in a declaration, they are still checked precisely, but the type information is of limited use to the compiler. and types are effective as long as the intersection can be canonicalized to a type that doesn't use and. For example:

```
(and fixnum unsigned-byte)
```

is fine, since it is the same as:

(integer 0 *most-positive-fixnum*)

but this type:

```
(and symbol (not (member :end)))
```

will not be fully understood by type interference since the and can't be removed by canonicalization.

Using any of these type specifiers in a type test with typep or typecase is fine, since as tests, these types can be translated into the and macro, the not function or a call to the satisfies predicate.

5.2.11 Type Style Recommendations

Python provides good support for some currently unconventional ways of using the Common Lisp type system. With Python, it is desirable to make declarations as precise as possible, but type inference also makes some declarations unnecessary. Here are some general guidelines for maximum robustness and efficiency:

- Declare the types of all function arguments and structure slots as precisely as possible (while avoiding not, and and satisfies). Put these declarations in during initial coding so that type assertions can find bugs for you during debugging.
- Use the member type specifier where there are a small number of possible symbol values, for example: (member :red :blue :green).
- Use the or type specifier in situations where the type is not certain, but there are only a few possibilities, for example: (or list vector).
- Declare integer types with the tightest bounds that you can, such as (integer 3 7).
- Define deftype or defstruct types before they are used. Definition after use is legal (producing no "undefined type" warnings), but type tests and structure operations will be compiled much less efficiently.
- Use the extensions:freeze-type declaration to speed up type testing for structure types which won't have new subtypes added later. See section 5.2.9, page 60
- In addition to declaring the array element type and simpleness, also declare the dimensions if they are fixed, for example:

```
(simple-array single-float (1024 1024))
```

This bounds information allows array indexing for multi-dimensional arrays to be compiled much more efficiently, and may also allow array bounds checking to be done at compile time. See section 5.10.3, page 83.

- Avoid use of the the declaration within expressions. Not only does it clutter the code, but it is also almost worthless under safe policies. If the need for an output type assertion is revealed by efficiency notes during tuning, then you can consider the, but it is preferable to constrain the argument types more, allowing the compiler to prove the desired result type.

- Don't bother declaring the type of let or other non-argument variables unless the type is non-obvious. If you declare function return types and structure slot types, then the type of a variable is often obvious both to the programmer and to the compiler. An important case where the type isn't obvious, and a declaration is appropriate, is when the value for a variable is pulled out of untyped structure (e.g., the result of car), or comes from some weakly typed function, such as read.

- Declarations are sometimes necessary for integer loop variables, since the compiler can't always prove that the value is of a good integer type. These declarations are best added during tuning, when an efficiency note indicates the need.

5.3 Type Inference

Type inference is the process by which the compiler tries to figure out the types of expressions and variables, given an inevitable lack of complete type information. Although Python does much more type inference than most Common Lisp compilers, remember that the more precise and comprehensive type declarations are, the more type inference will be able to do.

5.3.1 Variable Type Inference

The type of a variable is the union of the types of all the definitions. In the degenerate case of a let, the type of the variable is the type of the initial value. This inferred type is intersected with any declared type, and is then propagated to all the variable's references. The types of multiple-value-bind variables are similarly inferred from the types of the individual values of the values form.

If multiple type declarations apply to a single variable, then all the declarations must be correct; it is as though all the types were intersected producing a single and type specifier. In this example:

```
(defmacro my-dotimes ((var count) &body body)
  `(do ((,var 0 (1+ ,var)))
       ((>= ,var ,count))
     (declare (type (integer 0 *) ,var))
     ,@body))

(my-dotimes (i ...)
  (declare (fixnum i))
  ...)
```

the two declarations for i are intersected, so i is known to be a non-negative fixnum.

In practice, this type inference is limited to lets and local functions, since the compiler can't analyze all the calls to a global function. But type inference works well enough on local variables so that it is often unnecessary to declare the type of local variables. This is especially likely when function result types and structure slot types are declared. The main areas where type inference breaks down are:

- When the initial value of a variable is a untyped expression, such as (car x), and

- When the type of one of the variable's definitions is a function of the variable's current value, as in: (setq x (1+ x))

5.3.2 Local Function Type Inference

The types of arguments to local functions are inferred in the same was as any other local variable; the type is the union of the argument types across all the calls to the function, intersected with the declared type. If there are any assignments to the argument variables, the type of the assigned value is unioned in as well.

The result type of a local function is computed in a special way that takes tail recursion (see section 5.5, page 71) into consideration. The result type is the union of all possible return values that aren't tail-recursive calls. For example, Python will infer that the result type of this function is integer:

```
(defun ! (n res)
  (declare (integer n res))
  (if (zerop n)
      res
      (! (1- n) (* n res))))
```

Although this is a rather obvious result, it becomes somewhat less trivial in the presence of mutual tail recursion of multiple functions. Local function result type inference interacts with the mechanisms for ensuring proper tail recursion mentioned in section 5.6.5.

5.3.3 Global Function Type Inference

As described in section 5.2.6, a global function type (ftype) declaration places implicit type assertions on the call arguments, and also guarantees the type of the return value. So wherever a call to a declared function appears, there is no doubt as to the types of the arguments and return value. Furthermore, Python will infer a function type from the function's definition if there is no ftype declaration. Any type declarations on the argument variables are used as the argument types in the derived function type, and the compiler's best guess for the result type of the function is used as the result type in the derived function type.

This method of deriving function types from the definition implicitly assumes that functions won't be redefined at run-time. Consider this example:

```
(defun foo-p (x)
  (let ((res (and (consp x) (eq (car x) 'foo))))
    (format t "It is ~:[not ~;~]foo." res)))

(defun frob (it)
  (if (foo-p it)
      (setf (cadr it) 'yow!)
      (1+ it)))
```

Presumably, the programmer really meant to return res from foo-p, but he seems to have forgotten. When he tries to call do (frob (list 'foo nil)), frob will flame out when it tries to add to a cons. Realizing his error, he fixes foo-p and recompiles it. But when he retries his test case, he is baffled because the error is still there. What happened in this example is that Python proved that the result of foo-p is null, and then proceeded to optimize away the setf in frob.

Fortunately, in this example, the error is detected at compile time due to notes about unreachable code (see section 5.4.5, page 68.) Still, some users may not want to worry about this sort of problem during incremental development, so there is a variable to control deriving function types.

extensions:*derive-function-types* [*Variable*]

If true (the default), argument and result type information derived from compilation of defuns is used when compiling calls to that function. If false, only information from ftype proclamations will be used.

5.3.4 Operation Specific Type Inference

Many of the standard Common Lisp functions have special type inference procedures that determine the result type as a function of the argument types. For example, the result type of aref is the array element type. Here are some other examples of type inferences:

```
(logand x #xFF) ⇒ (unsigned-byte 8)

(+ (the (integer 0 12) x) (the (integer 0 1) y)) ⇒ (integer 0 13)

(ash (the (unsigned-byte 16) x) -8) ⇒ (unsigned-byte 8)
```

5.3.5 Dynamic Type Inference

Python uses flow analysis to infer types in dynamically typed programs. For example:

```
(ecase x
  (list (length x))
  ...)
```

Here, the compiler knows the argument to length is a list, because the call to length is only done when x is a list. The most significant efficiency effect of inference from assertions is usually in type check optimization.

Dynamic type inference has two inputs: explicit conditionals and implicit or explicit type assertions. Flow analysis propagates these constraints on variable type to any code that can be executed only after passing though the constraint. Explicit type constraints come from ifs where the test is either a lexical variable or a function of lexical variables and constants, where the function is either a type predicate, a numeric comparison or eq.

If there is an eq (or eql) test, then the compiler will actually substitute one argument for the other in the true branch. For example:

```
(when (eq x :yow!) (return x))
```

becomes:

```
(when (eq x :yow!) (return :yow!))
```

This substitution is done when one argument is a constant, or one argument has better type information than the other. This transformation reveals opportunities for constant folding or type-specific optimizations. If the test is against a constant, then the compiler can prove that the variable is not that constant value in the false branch, or (not (member :yow!)) in the example above. This can eliminate redundant tests, for example:

```
(if (eq x nil)
    ...
    (if x a b))
```

is transformed to this:

```
(if (eq x nil)
    ...
    a)
```

Variables appearing as if tests are interpreted as (not (eq *var* nil)) tests. The compiler also converts = into eql where possible. It is difficult to do inference directly on = since it does implicit coercions.

When there is an explicit < or > test on numeric variables, the compiler makes inferences about the ranges the variables can assume in the true and false branches. This is mainly useful when it proves that the values are small enough in magnitude to allow open-coding of arithmetic operations. For example, in many uses of dotimes with a fixnum repeat count, the compiler proves that fixnum arithmetic can be used.

Implicit type assertions are quite common, especially if you declare function argument types. Dynamic inference from implicit type assertions sometimes helps to disambiguate programs to a useful degree, but is most noticeable when it detects a dynamic type error. For example:

```
(defun foo (x)
  (+ (car x) x))
```

results in this warning:

```
In: DEFUN FOO
  (+ (CAR X) X)
==>
  X
Warning: Result is a LIST, not a NUMBER.
```

Note that Common Lisp's dynamic type checking semantics make dynamic type inference useful even in programs that aren't really dynamically typed, for example:

```
(+ (car x) (length x))
```

Here, x presumably always holds a list, but in the absence of a declaration the compiler cannot assume x is a list simply because list-specific operations are sometimes done on it. The compiler must consider the program to be dynamically typed until it proves otherwise. Dynamic type inference proves that the argument to length is always a list because the call to length is only done after the list-specific car operation.

5.3.6 Type Check Optimization

Python backs up its support for precise type checking by minimizing the cost of run-time type checking. This is done both through type inference and though optimizations of type checking itself.

Type inference often allows the compiler to prove that a value is of the correct type, and thus no type check is necessary. For example:

```
(defstruct foo a b c)
(defstruct link
  (foo (required-argument) :type foo)
  (next nil :type (or link null)))

(foo-a (link-foo x))
```

Here, there is no need to check that the result of link-foo is a foo, since it always is. Even when some type checks are necessary, type inference can often reduce the number:

```
(defun test (x)
  (let ((a (foo-a x))
        (b (foo-b x))
        (c (foo-c x)))
    ...))
```

In this example, only one (foo-p x) check is needed. This applies to a lesser degree in list operations, such as:

```
(if (eql (car x) 3) (cdr x) y)
```

Here, we only have to check that x is a list once.

Since Python recognizes explicit type tests, code that explicitly protects itself against type errors has little introduced overhead due to implicit type checking. For example, this loop compiles with no implicit checks checks for car and cdr:

```
(defun memq (e l)
  (do ((current l (cdr current)))
      ((atom current) nil)
    (when (eq (car current) e) (return current))))
```

Python reduces the cost of checks that must be done through an optimization called *complementing*. A complemented check for *type* is simply a check that the value is not of the type (not *type*). This is only interesting when something is known about the actual type, in which case we can test for the complement of (and *known-type* (not *type*)), or the difference between the known type and the assertion. An example:

```
(link-foo (link-next x))
```

Here, we change the type check for link-foo from a test for foo to a test for:

```
(not (and (or foo null) (not foo)))
```

or more simply (not null). This is probably the most important use of complementing, since the situation is fairly common, and a null test is much cheaper than a structure type test.

Here is a more complicated example that illustrates the combination of complementing with dynamic type inference:

```
(defun find-a (a x)
  (declare (type (or link null) x))
  (do ((current x (link-next current)))
      ((null current) nil)
    (let ((foo (link-foo current)))
      (when (eq (foo-a foo) a) (return foo)))))
```

This loop can be compiled with no type checks. The link test for link-foo and link-next is complemented to (not null), and then deleted because of the explicit null test. As before, no check is necessary for foo-a, since the link-foo is always a foo. This sort of situation shows how precise type checking combined with precise declarations can actually result in reduced type checking.

5.4 Source Optimization

This section describes source-level transformations that Python does on programs in an attempt to make them more efficient. Although source-level optimizations can make existing programs more efficient, the biggest advantage of this sort of optimization is that it makes it easier to write efficient programs. If a clean, straightforward implementation is can be transformed into an efficient one, then there is no need for tricky and dangerous hand optimization.

5.4.1 Let Optimization

The primary optimization of let variables is to delete them when they are unnecessary. Whenever the value of a let variable is a constant, a constant variable or a constant (local or non-notinline) function, the variable is deleted, and references to the variable are replaced with references to the constant expression. This is useful primarily in the expansion of macros or inline functions, where argument values are often constant in any given call, but are in general non-constant expressions that must be bound to preserve order of evaluation. Let variable optimization eliminates the need for macros to carefully avoid spurious bindings, and also makes inline functions just as efficient as macros.

A particularly interesting class of constant is a local function. Substituting for lexical variables that are bound to a function can substantially improve the efficiency of functional programming styles, for example:

```
(let ((a #'(lambda (x) (zow x))))
  (funcall a 3))
```

effectively transforms to:

```
(zow 3)
```

This transformation is done even when the function is a closure, as in:

```
(let ((a (let ((y (zug)))
           #'(lambda (x) (zow x y)))))
  (funcall a 3))
```

becoming:

```
(zow 3 (zug))
```

A constant variable is a lexical variable that is never assigned to, always keeping its initial value. Whenever possible, avoid setting lexical variables—instead bind a new variable to the new value. Except for loop variables, it is almost always possible to avoid setting lexical variables. This form:

```
(let ((x (f x)))
  ...)
```

is *more* efficient than this form:

```
(setq x (f x))
...
```

Setting variables makes the program more difficult to understand, both to the compiler and to the programmer. Python compiles assignments at least as efficiently as any other Common Lisp compiler, but most let optimizations are only done on constant variables.

Constant variables with only a single use are also optimized away, even when the initial value is not constant.[1] For example, this expansion of incf:

```
(let ((#:g3 (+ x 1)))
  (setq x #:G3))
```

becomes:

```
(setq x (+ x 1))
```

The type semantics of this transformation are more important than the elimination of the variable itself. Consider what happens when x is declared to be a fixnum; after the transformation, the compiler can compile the addition knowing that the result is a fixnum, whereas before the transformation the addition would have to allow for fixnum overflow.

Another variable optimization deletes any variable that is never read. This causes the initial value and any assigned values to be unused, allowing those expressions to be deleted if they have no side-effects.

Note that a let is actually a degenerate case of local call (see section 5.6.2, page 73), and that let optimization can be done on calls that weren't created by a let. Also, local call allows an applicative style of iteration that is totally assignment free.

5.4.2 Constant Folding

Constant folding is an optimization that replaces a call of constant arguments with the constant result of that call. Constant folding is done on all standard functions for which it is legal. Inline expansion allows folding of any constant parts of the definition, and can be done even on functions that have side-effects.

It is convenient to rely on constant folding when programming, as in this example:

```
(defconstant limit 42)

(defun foo ()
  (... (1- limit) ...))
```

Constant folding is also helpful when writing macros or inline functions, since it usually eliminates the need to write a macro that special-cases constant arguments.

Constant folding of a user defined function is enabled by the extensions:constant-function proclamation. In this example:

```
(declaim (ext:constant-function myfun))
(defun myexp (x y)
  (declare (single-float x y))
  (exp (* (log x) y)))

 ... (myexp 3.0 1.3) ...
```

The call to myexp is constant-folded to 4.1711674.

[1]The source transformation in this example doesn't represent the preservation of evaluation order implicit in the compiler's internal representation. Where necessary, the back end will reintroduce temporaries to preserve the semantics.

5.4.3 Unused Expression Elimination

If the value of any expression is not used, and the expression has no side-effects, then it is deleted. As with constant folding, this optimization applies most often when cleaning up after inline expansion and other optimizations. Any function declared an extensions:constant-function is also subject to unused expression elimination.

Note that Python will eliminate parts of unused expressions known to be side-effect free, even if there are other unknown parts. For example:

```
(let ((a (list (foo) (bar))))
  (if t
      (zow)
      (raz a)))
```

becomes:

```
(progn (foo) (bar))
(zow)
```

5.4.4 Control Optimization

The most important optimization of control is recognizing when an if test is known at compile time, then deleting the if, the test expression, and the unreachable branch of the if. This can be considered a special case of constant folding, although the test doesn't have to be truly constant as long as it is definitely not nil. Note also, that type inference propagates the result of an if test to the true and false branches, see section 5.3.5, page 63.

A related if optimization is this transformation:[2]

```
(if (if a b c) x y)
```

into:

```
(if a
    (if b x y)
    (if c x y))
```

The opportunity for this sort of optimization usually results from a conditional macro. For example:

```
(if (not a) x y)
```

is actually implemented as this:

```
(if (if a nil t) x y)
```

which is transformed to this:

```
(if a
    (if nil x y)
    (if t x y))
```

which is then optimized to this:

```
(if a y x)
```

Note that due to Python's internal representations, the if—if situation will be recognized even if other forms are wrapped around the inner if, like:

```
(if (let ((g ...))
      (loop
        ...
        (return (not g))
        ...))
    x y)
```

In Python, all the Common Lisp macros really are macros, written in terms of if, block and tagbody, so user-defined control macros can be just as efficient as the standard ones. Python emits basic blocks using a heuristic that minimizes the number of unconditional branches. The code in a tagbody will not be emitted in the order it appeared in the source, so there is no point in arranging the code to make control drop through to the target.

[2]Note that the code for x and y isn't actually replicated.

5.4.5 Unreachable Code Deletion

Python will delete code whenever it can prove that the code can never be executed. Code becomes unreachable when:

- An if is optimized away, or
- There is an explicit unconditional control transfer such as go or return-from, or
- The last reference to a local function is deleted (or there never was any reference.)

When code that appeared in the original source is deleted, the compiler prints a note to indicate a possible problem (or at least unnecessary code.) For example:

```
(defun foo ()
  (if t
      (write-line "True.")
      (write-line "False.")))
```

will result in this note:

```
In: DEFUN FOO
  (WRITE-LINE "False.")
Note: Deleting unreachable code.
```

It is important to pay attention to unreachable code notes, since they often indicate a subtle type error. For example:

```
(defstruct foo a b)

(defun lose (x)
  (let ((a (foo-a x))
        (b (if x (foo-b x) :none)))
    ...))
```

results in this note:

```
In: DEFUN LOSE
  (IF X (FOO-B X) :NONE)
==>
  :NONE
Note: Deleting unreachable code.
```

The :none is unreachable, because type inference knows that the argument to foo-a must be a foo, and thus can't be nil. Presumably the programmer forgot that x could be nil when he wrote the binding for a.

Here is an example with an incorrect declaration:

```
(defun count-a (string)
  (do ((pos 0 (position #\a string :start (1+ pos)))
       (count 0 (1+ count)))
      ((null pos) count)
    (declare (fixnum pos))))
```

This time our note is:

```
In: DEFUN COUNT-A
  (DO ((POS 0 #) (COUNT 0 #))
      ((NULL POS) COUNT)
    (DECLARE (FIXNUM POS)))
--> BLOCK LET TAGBODY RETURN-FROM PROGN
==>
  COUNT
Note: Deleting unreachable code.
```

The problem here is that pos can never be null since it is declared a fixnum.

It takes some experience with unreachable code notes to be able to tell what they are trying to say. In non-obvious cases, the best thing to do is to call the function in a way that should cause the unreachable code to be executed. Either you will get a type error, or you will find that there truly is no way for the code to be executed.

Not all unreachable code results in a note:

- A note is only given when the unreachable code textually appears in the original source. This prevents spurious notes due to the optimization of macros and inline functions, but sometimes also foregoes a note that would have been useful.
- Since accurate source information is not available for non-list forms, there is an element of heuristic in determining whether or not to give a note about an atom. Spurious notes may be given when a macro or inline function defines a variable that is also present in the calling function. Notes about nil and t are never given, since it is too easy to confuse these constants in expanded code with ones in the original source.
- Notes are only given about code unreachable due to control flow. There is no note when an expression is deleted because its value is unused, since this is a common consequence of other optimizations.

Somewhat spurious unreachable code notes can also result when a macro inserts multiple copies of its arguments in different contexts, for example:

```
(defmacro t-and-f (var form)
  `(if ,var ,form ,form))

(defun foo (x)
  (t-and-f x (if x "True." "False.")))
```

results in these notes:

```
In: DEFUN FOO
  (IF X "True." "False.")
==>
  "False."
Note: Deleting unreachable code.

==>
  "True."
Note: Deleting unreachable code.
```

It seems like it has deleted both branches of the if, but it has really deleted one branch in one copy, and the other branch in the other copy. Note that these messages are only spurious in not satisfying the intent of the rule that notes are only given when the deleted code appears in the original source; there is always *some* code being deleted when a unreachable code note is printed.

5.4.6 Multiple Values Optimization

Within a function, Python implements uses of multiple values particularly efficiently. Multiple values can be kept in arbitrary registers, so using multiple values doesn't imply stack manipulation and representation conversion. For example, this code:

```
(let ((a (if x (foo x) u))
      (b (if x (bar x) v)))
  ...)
```

is actually more efficient written this way:

```
(multiple-value-bind
    (a b)
    (if x
        (values (foo x) (bar x))
        (values u v))
  ...)
```

Also, see section 5.6.5, page 75 for information on how local call provides efficient support for multiple function return values.

5.4.7 Source to Source Transformation

The compiler implements a number of operation-specific optimizations as source-to-source transformations. You will often see unfamiliar code in error messages, for example:

```
(defun my-zerop () (zerop x))
```

gives this warning:

```
In: DEFUN MY-ZEROP
  (ZEROP X)
==>
  (= X 0)
Warning: Undefined variable: X
```

The original zerop has been transformed into a call to =. This transformation is indicated with the same ==> used to mark macro and function inline expansion. Although it can be confusing, display of the transformed source is important, since warnings are given with respect to the transformed source. This a more obscure example:

```
(defun foo (x) (logand 1 x))
```

gives this efficiency note:

```
In: DEFUN FOO
  (LOGAND 1 X)
==>
  (LOGAND C::Y C::X)
Note: Forced to do static-function Two-arg-and (cost 53).
      Unable to do inline fixnum arithmetic (cost 1) because:
      The first argument is a INTEGER, not a FIXNUM.
      etc.
```

Here, the compiler commuted the call to logand, introducing temporaries. The note complains that the *first* argument is not a fixnum, when in the original call, it was the second argument. To make things more confusing, the compiler introduced temporaries called c::x and c::y that are bound to y and 1, respectively.

You will also notice source-to-source optimizations when efficiency notes are enabled (see section 5.13, page 92.) When the compiler is unable to do a transformation that might be possible if there was more information, then an efficiency note is printed. For example, my-zerop above will also give this efficiency note:

```
In: DEFUN FOO
  (ZEROP X)
==>
  (= X 0)
Note: Unable to optimize because:
      Operands might not be the same type, so can't open code.
```

5.4.8 Style Recommendations

Source level optimization makes possible a clearer and more relaxed programming style:

- Don't use macros purely to avoid function call. If you want an inline function, write it as a function and declare it inline. It's clearer, less error-prone, and works just as well.

- Don't write macros that try to "optimize" their expansion in trivial ways such as avoiding binding variables for simple expressions. The compiler does these optimizations too, and is less likely to make a mistake.
- Make use of local functions (i.e., labels or flet) and tail-recursion in places where it is clearer. Local function call is faster than full call.
- Avoid setting local variables when possible. Binding a new let variable is at least as efficient as setting an existing variable, and is easier to understand, both for the compiler and the programmer.
- Instead of writing similar code over and over again so that it can be hand customized for each use, define a macro or inline function, and let the compiler do the work.

5.5 Tail Recursion

A call is tail-recursive if nothing has to be done after the the call returns, i.e. when the call returns, the returned value is immediately returned from the calling function. In this example, the recursive call to myfun is tail-recursive:

```
(defun myfun (x)
  (if (oddp (random x))
      (isqrt x)
      (myfun (1- x))))
```

Tail recursion is interesting because it is form of recursion that can be implemented much more efficiently than general recursion. In general, a recursive call requires the compiler to allocate storage on the stack at run-time for every call that has not yet returned. This memory consumption makes recursion unacceptably inefficient for representing repetitive algorithms having large or unbounded size. Tail recursion is the special case of recursion that is semantically equivalent to the iteration constructs normally used to represent repetition in programs. Because tail recursion is equivalent to iteration, tail-recursive programs can be compiled as efficiently as iterative programs.

So why would you want to write a program recursively when you can write it using a loop? Well, the main answer is that recursion is a more general mechanism, so it can express some solutions simply that are awkward to write as a loop. Some programmers also feel that recursion is a stylistically preferable way to write loops because it avoids assigning variables. For example, instead of writing:

```
(defun fun1 (x)
  something-that-uses-x)

(defun fun2 (y)
  something-that-uses-y)

(do ((x something (fun2 (fun1 x))))
    (nil))
```

You can write:

```
(defun fun1 (x)
  (fun2 something-that-uses-x))

(defun fun2 (y)
  (fun1 something-that-uses-y))

(fun1 something)
```

The tail-recursive definition is actually more efficient, in addition to being (arguably) clearer. As the number of functions and the complexity of their call graph increases, the simplicity of using recursion becomes compelling. Consider the advantages of writing a large finite-state machine with separate tail-recursive functions instead of using a single huge prog.

It helps to understand how to use tail recursion if you think of a tail-recursive call as a psetq that assigns the argument values to the called function's variables, followed by a go to the start of the called function. This makes clear an inherent efficiency advantage of tail-recursive call: in addition to not having to allocate a stack frame, there is no need to prepare for the call to return (e.g., by computing a return PC.)

Is there any disadvantage to tail recursion? Other than an increase in efficiency, the only way you can tell that a call has been compiled tail-recursively is if you use the debugger. Since a tail-recursive call has no stack frame, there is no way the debugger can print out the stack frame representing the call. The effect is that backtrace will not show some calls that would have been displayed in a non-tail-recursive implementation. In practice, this is not as bad as it sounds—in fact it isn't really clearly worse, just different. See section 3.3.5, page 28 for information about the debugger implications of tail recursion.

In order to ensure that tail-recursion is preserved in arbitrarily complex calling patterns across separately compiled functions, the compiler must compile any call in a tail-recursive position as a tail-recursive call. This is done regardless of whether the program actually exhibits any sort of recursive calling pattern. In this example, the call to fun2 will always be compiled as a tail-recursive call:

```
(defun fun1 (x)
  (fun2 x))
```

So tail recursion doesn't necessarily have anything to do with recursion as it is normally thought of. See section 5.6.4, page 74 for more discussion of using tail recursion to implement loops.

5.5.1 Tail Recursion Exceptions

Although Python is claimed to be "properly" tail-recursive, some might dispute this, since there are situations where tail recursion is inhibited:

- When the call is enclosed by a special binding, or
- When the call is enclosed by a catch or unwind-protect, or
- When the call is enclosed by a block or tagbody and the block name or go tag has been closed over.

These dynamic extent binding forms inhibit tail recursion because they allocate stack space to represent the binding. Shallow-binding implementations of dynamic scoping also require cleanup code to be evaluated when the scope is exited.

5.6 Local Call

Python supports two kinds of function call: full call and local call. Full call is the standard calling convention; its late binding and generality make Common Lisp what it is, but create unavoidable overheads. When the compiler can compile the calling function and the called function simultaneously, it can use local call to avoid some of the overhead of full call. Local call is really a collection of compilation strategies. If some aspect of call overhead is not needed in a particular local call, then it can be omitted. In some cases, local call can be totally free. Local call provides two main advantages to the user:

- Local call makes the use of the lexical function binding forms flet and labels much more efficient. A local call is always faster than a full call, and in many cases is much faster.
- Local call is a natural approach to *block compilation*, a compilation technique that resolves function references at compile time. Block compilation speeds function call, but increases compilation times and prevents function redefinition.

5.6.1 Self-Recursive Calls

Local call is used when a function defined by defun calls itself. For example:

```
(defun fact (n)
  (if (zerop n)
      1
      (* n (fact (1- n)))))
```

This use of local call speeds recursion, but can also complicate debugging, since trace will only show the first call to fact, and not the recursive calls. This is because the recursive calls directly jump to the start of the function, and don't indirect through the symbol-function. Self-recursive local call is inhibited when the :block-compile argument to compile-file is nil (see section 5.7.3, page 76.)

5.6.2 Let Calls

Because local call avoids unnecessary call overheads, the compiler internally uses local call to implement some macros and special forms that are not normally thought of as involving a function call. For example, this let:

```
(let ((a (foo))
      (b (bar)))
  ...)
```

is internally represented as though it was macroexpanded into:

```
(funcall #'(lambda (a b)
             ...)
         (foo)
         (bar))
```

This implementation is acceptable because the simple cases of local call (equivalent to a let) result in good code. This doesn't make let any more efficient, but does make local calls that are semantically the same as let much more efficient than full calls. For example, these definitions are all the same as far as the compiler is concerned:

```
(defun foo ()
  ...some other stuff...
  (let ((a something))
    ...some stuff...))

(defun foo ()
  (flet ((localfun (a)
           ...some stuff...))
    ...some other stuff...
    (localfun something)))

(defun foo ()
  (let ((funvar #'(lambda (a)
                    ...some stuff...)))
    ...some other stuff...
    (funcall funvar something)))
```

Although local call is most efficient when the function is called only once, a call doesn't have to be equivalent to a let to be more efficient than full call. All local calls avoid the overhead of argument count checking and keyword argument parsing, and there are a number of other advantages that apply in many common situations. See section 5.4.1, page 65 for a discussion of the optimizations done on let calls.

5.6.3 Closures

Local call allows for much more efficient use of closures, since the closure environment doesn't need to be allocated on the heap, or even stored in memory at all. In this example, there is no penalty for localfun referencing a and b:

```
(defun foo (a b)
  (flet ((localfun (x)
           (1+ (* a b x))))
    (if (= a b)
        (localfun (- x))
        (localfun x))))
```

In local call, the compiler effectively passes closed-over values as extra arguments, so there is no need for you to "optimize" local function use by explicitly passing in lexically visible values. Closures may also be subject to let optimization (see section 5.4.1, page 65.)

Note: indirect value cells are currently always allocated on the heap when a variable is both assigned to (with setq or setf) and closed over, regardless of whether the closure is a local function or not. This is another reason to avoid setting variables when you don't have to.

5.6.4 Local Tail Recursion

Tail-recursive local calls are particularly efficient, since they are in effect an assignment plus a control transfer. Scheme programmers write loops with tail-recursive local calls, instead of using the imperative go and setq. This has not caught on in the Common Lisp community, since conventional Common Lisp compilers don't implement local call. In Python, users can choose to write loops such as:

```
(defun ! (n)
  (labels ((loop (n total)
             (if (zerop n)
                 total
                 (loop (1- n) (* n total)))))
    (loop n 1)))
```

extensions:iterate *name* ({*(var initial-value)*}*) {*declaration*}* {*form*}* [*Macro*]

This macro provides syntactic sugar for using labels to do iteration. It creates a local function *name* with the specified *var*s as its arguments and the *declaration*s and *form*s as its body. This function is then called with the *initial-values*, and the result of the call is return from the macro.

Here is our factorial example rewritten using iterate:

```
(defun ! (n)
  (iterate loop
           ((n n)
           (total 1))
    (if (zerop n)
      total
      (loop (1- n) (* n total)))))
```

The main advantage of using iterate over do is that iterate naturally allows stepping to be done differently depending on conditionals in the body of the loop. iterate can also be used to implement algorithms that aren't really iterative by simply doing a non-tail call. For example, the standard recursive definition of factorial can be written like this:

```
(iterate fact
         ((n n))
  (if (zerop n)
      1
      (* n (fact (1- n)))))
```

5.6.5 Return Values

One of the more subtle costs of full call comes from allowing arbitrary numbers of return values. This overhead can be avoided in local calls to functions that always return the same number of values. For efficiency reasons (as well as stylistic ones), you should write functions so that they always return the same number of values. This may require passing extra nil arguments to values in some cases, but the result is more efficient, not less so.

When efficiency notes are enabled (see section 5.13, page 92), and the compiler wants to use known values return, but can't prove that the function always returns the same number of values, then it will print a note like this:

```
In: DEFUN GRUE
  (DEFUN GRUE (X) (DECLARE (FIXNUM X)) (COND (# #) (# NIL) (T #)))
Note: Return type not fixed values, so can't use known return convention:
  (VALUES (OR (INTEGER -536870912 -1) NULL) &REST T)
```

In order to implement proper tail recursion in the presence of known values return (see section 5.5, page 71), the compiler sometimes must prove that multiple functions all return the same number of values. When this can't be proven, the compiler will print a note like this:

```
In: DEFUN BLUE
  (DEFUN BLUE (X) (DECLARE (FIXNUM X)) (COND (# #) (# #) (# #) (T #)))
Note: Return value count mismatch prevents known return from
      these functions:
  BLUE
  SNOO
```

See section 5.11.10, page 90 for the interaction between local call and the representation of numeric types.

5.7 Block Compilation

Block compilation allows calls to global functions defined by defun to be compiled as local calls. The function call can be in a different top-level form than the defun, or even in a different file.

In addition, block compilation allows the declaration of the *entry points* to the block compiled portion. An entry point is any function that may be called from outside of the block compilation. If a function is not an entry point, then it can be compiled more efficiently, since all calls are known at compile time. In particular, if a function is only called in one place, then it will be let converted. This effectively inline expands the function, but without the code duplication that results from defining the function normally and then declaring it inline.

The main advantage of block compilation is that it it preserves efficiency in programs even when (for readability and syntactic convenience) they are broken up into many small functions. There is absolutely no overhead for calling a non-entry point function that is defined purely for modularity (i.e. called only in one place.)

Block compilation also allows the use of non-descriptor arguments and return values in non-trivial programs (see section 5.11.10, page 90).

5.7.1 Block Compilation Semantics

The effect of block compilation can be envisioned as the compiler turning all the defuns in the block compilation into a single labels form:

```
(declaim (start-block fun1 fun3))

(defun fun1 ()
  ...)

(defun fun2 ()
  ...
  (fun1)
  ...)
```

```
(defun fun3 (x)
  (if x
      (fun1)
      (fun2)))

(declaim (end-block))
```

becomes:

```
(labels ((fun1 ()
           ...)
         (fun2 ()
           ...
           (fun1)
           ...)
         (fun3 (x)
           (if x
               (fun1)
               (fun2))))
  (setf (fdefinition 'fun1) #'fun1)
  (setf (fdefinition 'fun3) #'fun3))
```

Calls between the block compiled functions are local calls, so changing the global definition of fun1 will have no effect on what fun2 does; fun2 will keep calling the old fun1.

The entry points fun1 and fun3 are still installed in the symbol-function as the global definitions of the functions, so a full call to an entry point works just as before. However, fun2 is not an entry point, so it is not globally defined. In addition, fun2 is only called in one place, so it will be let converted.

5.7.2 Block Compilation Declarations

The extensions:start-block and extensions:end-block declarations allow fine-grained control of block compilation. These declarations are only legal as a global declarations (declaim or proclaim).

The start-block declaration has this syntax:

(start-block $\{$*entry-point-name*$\}^*$)

When processed by the compiler, this declaration marks the start of block compilation, and specifies the entry points to that block. If no entry points are specified, then *all* functions are made into entry points. If already block compiling, then the compiler ends the current block and starts a new one.

The end-block declaration has no arguments:

```
(end-block)
```

The end-block declaration ends a block compilation unit without starting a new one. This is useful mainly when only a portion of a file is worth block compiling.

5.7.3 Compiler Arguments

The :block-compile and :entry-points arguments to extensions:compile-from-stream and compile-file (page 38) provide overall control of block compilation, and allow block compilation without requiring modification of the program source.

There are three possible values of the :block-compile argument:

nil Do no compile-time resolution of global function names, not even for self-recursive calls. This inhibits any start-block declarations appearing in the file, allowing all functions to be incrementally redefined.

t
: Start compiling in block compilation mode. This is mainly useful for block compiling small files that contain no start-block declarations. See also the :entry-points argument.

:specified
: Start compiling in form-at-a-time mode, but exploit start-block declarations and compile self-recursive calls as local calls. Normally :specified is the default for this argument (see *block-compile-default* (page 77).)

The :entry-points argument can be used in conjunction with :block-compile t to specify the entry-points to a block-compiled file. If not specified or nil, all global functions will be compiled as entry points. When :block-compile is not t, this argument is ignored.

block-compile-default [*Variable*]

This variable determines the default value for the :block-compile argument to compile-file and compile-from-stream. The initial value of this variable is :specified, but nil is sometimes useful for totally inhibiting block compilation.

5.7.4 Practical Difficulties

The main problem with block compilation is that the compiler uses large amounts of memory when it is block compiling. This places an upper limit on the amount of code that can be block compiled as a unit. To make best use of block compilation, it is necessary to locate the parts of the program containing many internal calls, and then add the appropriate start-block declarations. When writing new code, it is a good idea to put in block compilation declarations from the very beginning, since writing block declarations correctly requires accurate knowledge of the program's function call structure. If you want to initially develop code with full incremental redefinition, you can compile with *block-compile-default* (page 77) set to nil.

Note if a defun appears in a non-null lexical environment, then calls to it cannot be block compiled.

Unless files are very small, it is probably impractical to block compile multiple files as a unit by specifying a list of files to compile-file. Semi-inline expansion (see section 5.8.2, page 80) provides another way to extend block compilation across file boundaries.

5.7.5 Context Declarations

CMUCL has a context-sensitive declaration mechanism which is useful because it allows flexible control of the compilation policy in large systems without requiring changes to the source files. The primary use of this feature is to allow the exported interfaces of a system to be compiled more safely than the system internals. The context used is the name being defined and the kind of definition (function, macro, etc.)

The :context-declarations option to with-compilation-unit (page 40) has dynamic scope, affecting all compilation done during the evaluation of the body. The argument to this option should evaluate to a list of lists of the form:

(*context-spec* {*declare-form*}$^+$)

In the indicated context, the specified declare forms are inserted at the head of each definition. The declare forms for all contexts that match are appended together, with earlier declarations getting precedence over later ones. A simple example:

```
:context-declarations
'((:external (declare (optimize (safety 2)))))
```

This will cause all functions that are named by external symbols to be compiled with safety 2.

The full syntax of context specs is:

:internal, :external
: True if the symbol is internal (external) in its home package.

:uninterned
: True if the symbol has no home package.

(:package {*package-name*}*)
: True if the symbol's home package is in any of the named packages (false if uninterned.)

:anonymous
: True if the function doesn't have any interesting name (not defmacro, defun, labels or flet).

:macro, :function
: :macro is a global (defmacro) macro. :function is anything else.

:local, :global
: :local is a labels or flet. :global is anything else.

(:or {*context-spec*}*)
: True when any supplied *context-spec* is true.

(:and {*context-spec*}*)
: True only when all supplied *context-spec*s are true.

(:not {*context-spec*}*)
: True when *context-spec* is false.

(:member {*name*}*)
: True when the defined name is one of these names (equal test.)

(:match {*pattern*}*)
: True when any of the patterns is a substring of the name. The name is wrapped with $'s, so "$FOO" matches names beginning with "FOO", etc.

5.7.6 Context Declaration Example

Here is a more complex example of with-compilation-unit options:

```
:optimize '(optimize (speed 2) (space 2) (inhibit-warnings 2)
                     (debug 1) (safety 0))
:optimize-interface '(optimize-interface (safety 1) (debug 1))
:context-declarations
'(((:or :external (:and (:match "%") (:match "SET")))
   (declare (optimize-interface (safety 2))))
  ((:or (:and :external :macro)
        (:match "$PARSE-"))
   (declare (optimize (safety 2)))))
```

The optimize and extensions:optimize-interface declarations (see section 4.7.1, page 50) set up the global compilation policy. The bodies of functions are to be compiled completely unsafe (safety 0), but argument count and weakened argument type checking is to be done when a function is called (speed 2 safety 1).

The first declaration specifies that all functions that are external or whose names contain both "%" and "SET" are to be compiled compiled with completely safe interfaces (safety 2). The reason for this particular :match rule is that setf inverse functions in this system tend to have both strings in their name somewhere. We want setf inverses to be safe because they are implicitly called by users even though their name is not exported.

The second declaration makes external macros or functions whose names start with "PARSE-" have safe bodies (as well as interfaces). This is desirable because a syntax error in a macro may cause a type error inside the body. The :match rule is used because macros often have auxiliary functions whose names begin with this string.

This particular example is used to build part of the standard CMUCL system. Note however, that context declarations must be set up according to the needs and coding conventions of a particular system; different parts of CMUCL are compiled with different context declarations, and your system will probably need its own declarations. In particular, any use of the :match option depends on naming conventions used in coding.

5.8 Inline Expansion

Python can expand almost any function inline, including functions with keyword arguments. The only restrictions are that keyword argument keywords in the call must be constant, and that global function definitions (defun) must be done in a null lexical environment (not nested in a let or other binding form.) Local functions (flet) can be inline expanded in any environment. Combined with Python's source-level optimization, inline expansion can be used for things that formerly required macros for efficient implementation. In Python, macros don't have any efficiency advantage, so they need only be used where a macro's syntactic flexibility is required.

Inline expansion is a compiler optimization technique that reduces the overhead of a function call by simply not doing the call: instead, the compiler effectively rewrites the program to appear as though the definition of the called function was inserted at each call site. In Common Lisp, this is straightforwardly expressed by inserting the lambda corresponding to the original definition:

```
(proclaim '(inline my-1+))
(defun my-1+ (x) (+ x 1))

(my-1+ someval) ⇒ ((lambda (x) (+ x 1)) someval)
```

When the function expanded inline is large, the program after inline expansion may be substantially larger than the original program. If the program becomes too large, inline expansion hurts speed rather than helping it, since hardware resources such as physical memory and cache will be exhausted. Inline expansion is called for:

- When profiling has shown that a relatively simple function is called so often that a large amount of time is being wasted in the calling of that function (as opposed to running in that function.) If a function is complex, it will take a long time to run relative the time spent in call, so the speed advantage of inline expansion is diminished at the same time the space cost of inline expansion is increased. Of course, if a function is rarely called, then the overhead of calling it is also insignificant.
- With functions so simple that they take less space to inline expand than would be taken to call the function (such as my-1+ above.) It would require intimate knowledge of the compiler to be certain when inline expansion would reduce space, but it is generally safe to inline expand functions whose definition is a single function call, or a few calls to simple Common Lisp functions.

In addition to this speed/space tradeoff from inline expansion's avoidance of the call, inline expansion can also reveal opportunities for optimization. Python's extensive source-level optimization can make use of context information from the caller to tremendously simplify the code resulting from the inline expansion of a function.

The main form of caller context is local information about the actual argument values: what the argument types are and whether the arguments are constant. Knowledge about argument types can eliminate run-time type tests (e.g., for generic arithmetic.) Constant arguments in a call provide opportunities for constant folding optimization after inline expansion.

A hidden way that constant arguments are often supplied to functions is through the defaulting of unsupplied optional or keyword arguments. There can be a huge efficiency advantage to inline expanding functions that have complex keyword-based interfaces, such as this definition of the member function:

```
(proclaim '(inline member))
(defun member (item list &key
                     (key #'identity)
                     (test #'eql testp)
                     (test-not nil notp))
  (do ((list list (cdr list)))
      ((null list) nil)
    (let ((car (car list)))
      (if (cond (testp
                 (funcall test item (funcall key car)))
                (notp
                 (not (funcall test-not item (funcall key car))))
```

```
                (t
                 (funcall test item (funcall key car))))
           (return list)))))
```

After inline expansion, this call is simplified to the obvious code:

```
(member a l :key #'foo-a :test #'char=) ⇒

(do ((list list (cdr list)))
    ((null list) nil)
  (let ((car (car list)))
    (if (char= item (foo-a car))
        (return list))))
```

In this example, there could easily be more than an order of magnitude improvement in speed. In addition to eliminating the original call to member, inline expansion also allows the calls to char= and foo-a to be open-coded. We go from a loop with three tests and two calls to a loop with one test and no calls.

See section 5.4, page 65 for more discussion of source level optimization.

5.8.1 Inline Expansion Recording

Inline expansion requires that the source for the inline expanded function to be available when calls to the function are compiled. The compiler doesn't remember the inline expansion for every function, since that would take an excessive about of space. Instead, the programmer must tell the compiler to record the inline expansion before the definition of the inline expanded function is compiled. This is done by globally declaring the function inline before the function is defined, by using the inline and extensions:maybe-inline (see section 5.8.3, page 81) declarations.

In addition to recording the inline expansion of inline functions at the time the function is compiled, compile-file also puts the inline expansion in the output file. When the output file is loaded, the inline expansion is made available for subsequent compilations; there is no need to compile the definition again to record the inline expansion.

If a function is declared inline, but no expansion is recorded, then the compiler will give an efficiency note like:

```
Note: MYFUN is declared inline, but has no expansion.
```

When you get this note, check that the inline declaration and the definition appear before the calls that are to be inline expanded. This note will also be given if the inline expansion for a defun could not be recorded because the defun was in a non-null lexical environment.

5.8.2 Semi-Inline Expansion

Python supports *semi-inline* functions. Semi-inline expansion shares a single copy of a function across all the calls in a component by converting the inline expansion into a local function (see section 5.6, page 72.) This takes up less space when there are multiple calls, but also provides less opportunity for context dependent optimization. When there is only one call, the result is identical to normal inline expansion. Semi-inline expansion is done when the space optimization quality is 0, and the function has been declared extensions:maybe-inline.

This mechanism of inline expansion combined with local call also allows recursive functions to be inline expanded. If a recursive function is declared inline, calls will actually be compiled semi-inline. Although recursive functions are often so complex that there is little advantage to semi-inline expansion, it can still be useful in the same sort of cases where normal inline expansion is especially advantageous, i.e. functions where the calling context can help a lot.

5.8.3 The Maybe-Inline Declaration

The extensions:maybe-inline declaration is a CMUCL extension. It is similar to inline, but indicates that inline expansion may sometimes be desirable, rather than saying that inline expansion should almost always be done. When used in a global declaration, extensions:maybe-inline causes the expansion for the named functions to be recorded, but the functions aren't actually inline expanded unless space is 0 or the function is eventually (perhaps locally) declared inline.

Use of the extensions:maybe-inline declaration followed by the defun is preferable to the standard idiom of:

```
(proclaim '(inline myfun))
(defun myfun () ...)
(proclaim '(notinline myfun))

;;; Any calls to myfun here are not inline expanded.

(defun somefun ()
  (declare (inline myfun))
  ;;
  ;; Calls to myfun here are inline expanded.
  ...)
```

The problem with using notinline in this way is that in Common Lisp it does more than just suppress inline expansion, it also forbids the compiler to use any knowledge of myfun until a later inline declaration overrides the notinline. This prevents compiler warnings about incorrect calls to the function, and also prevents block compilation.

The extensions:maybe-inline declaration is used like this:

```
(proclaim '(extensions:maybe-inline myfun))
(defun myfun () ...)

;;; Any calls to myfun here are not inline expanded.

(defun somefun ()
  (declare (inline myfun))
  ;;
  ;; Calls to myfun here are inline expanded.
  ...)

(defun someotherfun ()
  (declare (optimize (space 0)))
  ;;
  ;; Calls to myfun here are expanded semi-inline.
  ...)
```

In this example, the use of extensions:maybe-inline causes the expansion to be recorded when the defun for somefun is compiled, and doesn't waste space through doing inline expansion by default. Unlike notinline, this declaration still allows the compiler to assume that the known definition really is the one that will be called when giving compiler warnings, and also allows the compiler to do semi-inline expansion when the policy is appropriate.

When the goal is merely to control whether inline expansion is done by default, it is preferable to use extensions:maybe-inline rather than notinline. The notinline declaration should be reserved for those special occasions when a function may be redefined at run-time, so the compiler must be told that the obvious definition of a function is not necessarily the one that will be in effect at the time of the call.

5.9 Byte Coded Compilation

Python supports byte compilation to reduce the size of Lisp programs by allowing functions to be compiled more compactly. Byte compilation provides an extreme speed/space tradeoff: byte code is typically six times more compact than native code, but runs fifty times (or more) slower. This is about ten times faster than the standard interpreter, which is itself considered fast in comparison to other Common Lisp interpreters.

Large Lisp systems (such as CMUCL itself) often have large amounts of user-interface code, compile-time (macro) code, debugging code, or rarely executed special-case code. This code is a good target for byte compilation: very little time is spent running in it, but it can take up quite a bit of space. Straight-line code with many function calls is much more suitable than inner loops.

When byte-compiling, the compiler compiles about twice as fast, and can produce a hardware independent object file ('.bytef' type.) This file can be loaded like a normal fasl file on any implementation of CMUCL with the same byte-ordering.

The decision to byte compile or native compile can be done on a per-file or per-code-object basis. The :byte-compile argument to compile-file (page 38) has these possible values:

nil Don't byte compile anything in this file.

t Byte compile everything in this file and produce a processor-independent '.bytef' file.

:maybe Produce a normal fasl file, but byte compile any functions for which the speed optimization quality is 0 and the debug quality is not greater than 1.

extensions:*byte-compile-top-level* [*Variable*]

If this variable is true (the default) and the :byte-compile argument to compile-file is :maybe, then byte compile top-level code (code outside of any defun, defmethod, etc.)

extensions:*byte-compile-default* [*Variable*]

This variable determines the default value for the :byte-compile argument to compile-file, initially :maybe.

5.10 Object Representation

A somewhat subtle aspect of writing efficient Common Lisp programs is choosing the correct data structures so that the underlying objects can be implemented efficiently. This is partly because of the need for multiple representations for a given value (see section 5.11.2, page 85), but is also due to the sheer number of object types that Common Lisp has built in. The number of possible representations complicates the choice of a good representation because semantically similar objects may vary in their efficiency depending on how the program operates on them.

5.10.1 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 three percent of *Common Lisp: The Language II*. The language has grown since Lisp 1.5—new data types supersede lists for many purposes.

5.10.2 Structure Representation

One of the best ways of building complex data structures is to define appropriate structure types using defstruct. In Python, access of structure slots is always at least as fast as list or vector access, and is usually faster. In comparison to a list representation of a tuple, structures also have a space advantage.

Even if structures weren't more efficient than other representations, structure use would still be attractive because programs that use structures in appropriate ways are much more maintainable and robust than programs written using only lists. For example:

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

could have been written using structures in this way:

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

The second version is more maintainable because it is easier to understand what it is doing. It is more robust because structures accesses are type checked. An `astronaut` will never be confused with a `beverage`, and the result of `beverage-flavor` is always a flavor. See sections 5.2.8 and 5.2.9 for more information about structure types. See section 5.3, page 61 for a number of examples that make clear the advantages of structure typing.

Note that the structure definition should be compiled before any uses of its accessors or type predicate so that these function calls can be efficiently open-coded.

5.10.3 Arrays

Arrays are often the most efficient representation for collections of objects because:

- Array representations are often the most compact. An array is always more compact than a list containing the same number of elements.
- Arrays allow fast constant-time access.
- Arrays are easily destructively modified, which can reduce consing.
- Array element types can be specialized, which reduces both overall size and consing (see section 5.11.8, page 89.)

Access of arrays that are not of type `simple-array` is less efficient, so declarations are appropriate when an array is of a simple type like `simple-string` or `simple-bit-vector`. Arrays are almost always simple, but the compiler may not be able to prove simpleness at every use. The only way to get a non-simple array is to use the `:displaced-to`, `:fill-pointer` or `adjustable` arguments to `make-array`. If you don't use these hairy options, then arrays can always be declared to be simple.

Because of the many specialized array types and the possibility of non-simple arrays, array access is much like generic arithmetic (see section 5.11.4, page 86). In order for array accesses to be efficiently compiled, the element type and simpleness of the array must be known at compile time. If there is inadequate information, the compiler is forced to call a generic array access routine. You can detect inefficient array accesses by enabling efficiency notes, see section 5.13, page 92.

5.10.4 Vectors

Vectors (one dimensional arrays) are particularly useful, since in addition to their obvious array-like applications, they are also well suited to representing sequences. In comparison to a list representation, vectors are faster to access and take up between two and sixty-four times less space (depending on the element type.) As with arbitrary arrays, the compiler needs to know that vectors are not complex, so you should use `simple-string` in preference to `string`, etc.

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 functions you should probably be using a vector.

5.10.5 Bit-Vectors

Another thing that lists have been used for is set manipulation. In applications where there is a known, reasonably small universe of items bit-vectors can 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-vector`s are efficient, since they operate on a word at a time.

5.10.6 Hashtables

Hashtables are an efficient and general mechanism for maintaining associations such as the association between an object and its name. Although hashtables are usually the best way to maintain associations, efficiency and style considerations sometimes favor the use of an association list (a-list).

assoc is fairly fast when the *test* argument is eq or eql and 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, 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. 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 Python. 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 with setf. With an alist, one has to 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.

Historically, symbol property lists were often used for global name associations. Property lists provide an awkward and error-prone combination of name association and record structure. If you must use the property list, please store all the related values in a single structure under a single property, rather than using many properties. This makes access more efficient, and also adds a modicum of typing and abstraction. See section 5.2, page 56 for information on types in CMUCL.

5.11 Numbers

Numbers are interesting because numbers are one of the few Common Lisp data types that have direct support in conventional hardware. If a number can be represented in the way that the hardware expects it, then there is a big efficiency advantage.

Using hardware representations is problematical in Common Lisp due to dynamic typing (where the type of a value may be unknown at compile time.) It is possible to compile code for statically typed portions of a Common Lisp program with efficiency comparable to that obtained in statically typed languages such as C, but not all Common Lisp implementations succeed. There are two main barriers to efficient numerical code in Common Lisp:

- The compiler must prove that the numerical expression is in fact statically typed, and
- The compiler must be able to somehow reconcile the conflicting demands of the hardware mandated number representation with the Common Lisp requirements of dynamic typing and garbage-collecting dynamic storage allocation.

Because of its type inference (see section 5.3, page 61) and efficiency notes (see section 5.13, page 92), Python is better than conventional Common Lisp compilers at ensuring that numerical expressions are statically typed. Python also goes somewhat farther than existing compilers in the area of allowing native machine number representations in the presence of garbage collection.

5.11.1 Descriptors

Common Lisp's dynamic typing requires that it be possible to represent any value with a fixed length object, known as a *descriptor*. This fixed-length requirement is implicit in features such as:

- Data types (like simple-vector) that can contain any type of object, and that can be destructively modified to contain different objects (of possibly different types.)
- Functions that can be called with any type of argument, and that can be redefined at run time.

In order to save space, a descriptor is invariably represented as a single word. Objects that can be directly represented in the descriptor itself are said to be *immediate*. Descriptors for objects larger than one word are in reality pointers to the memory actually containing the object.

Representing objects using pointers has two major disadvantages:

- The memory pointed to must be allocated on the heap, so it must eventually be freed by the garbage collector. Excessive heap allocation of objects (or "consing") is inefficient in several ways. See section 5.12.2, page 90.
- Representing an object in memory requires the compiler to emit additional instructions to read the actual value in from memory, and then to write the value back after operating on it.

The introduction of garbage collection makes things even worse, since the garbage collector must be able to determine whether a descriptor is an immediate object or a pointer. This requires that a few bits in each descriptor be dedicated to the garbage collector. The loss of a few bits doesn't seem like much, but it has a major efficiency implication—objects whose natural machine representation is a full word (integers and single-floats) cannot have an immediate representation. So the compiler is forced to use an unnatural immediate representation (such as fixnum) or a natural pointer representation (with the attendant consing overhead.)

5.11.2 Non-Descriptor Representations

From the discussion above, we can see that the standard descriptor representation has many problems, the worst being number consing. Common Lisp compilers try to avoid these descriptor efficiency problems by using *non-descriptor* representations. A compiler that uses non-descriptor representations can compile this function so that it does no number consing:

```
(defun multby (vec n)
  (declare (type (simple-array single-float (*)) vec)
           (single-float n))
  (dotimes (i (length vec))
    (setf (aref vec i)
          (* n (aref vec i)))))
```

If a descriptor representation were used, each iteration of the loop might cons two floats and do three times as many memory references.

As its negative definition suggests, the range of possible non-descriptor representations is large. The performance improvement from non-descriptor representation depends upon both the number of types that have non-descriptor representations and the number of contexts in which the compiler is forced to use a descriptor representation.

Many Common Lisp compilers support non-descriptor representations for float types such as single-float and double-float (section 5.11.7.) Python adds support for full word integers (see section 5.11.6, page 87), characters (see section 5.11.11, page 90) and system-area pointers (unconstrained pointers, see section 6.5, page 102.) Many Common Lisp compilers support non-descriptor representations for variables (section 5.11.3) and array elements (section 5.11.8.) Python adds support for non-descriptor arguments and return values in local call (see section 5.11.10, page 90) and structure slots (see section 5.11.9, page 89).

5.11.3 Variables

In order to use a non-descriptor representation for a variable or expression intermediate value, the compiler must be able to prove that the value is always of a particular type having a non-descriptor representation. Type inference (see section 5.3, page 61) often needs some help from user-supplied declarations. The best kind of type declaration is a variable type declaration placed at the binding point:

```
(let ((x (car l)))
  (declare (single-float x))
  ...)
```

Use of the, or of variable declarations not at the binding form is insufficient to allow non-descriptor representation of the variable—with these declarations it is not certain that all values of the variable are of the right type. It is sometimes useful to introduce a gratuitous binding that allows the compiler to change to a non-descriptor representation, like:

```
(etypecase x
  ((signed-byte 32)
   (let ((x x))
     (declare (type (signed-byte 32) x))
     ...))
  ...)
```

The declaration on the inner x is necessary here due to a phase ordering problem. Although the compiler will eventually prove that the outer x is a (signed-byte 32) within that etypecase branch, the inner x would have been optimized away by that time. Declaring the type makes let optimization more cautious.

Note that storing a value into a global (or special) variable always forces a descriptor representation. Wherever possible, you should operate only on local variables, binding any referenced globals to local variables at the beginning of the function, and doing any global assignments at the end.

Efficiency notes signal use of inefficient representations, so programmer's needn't continuously worry about the details of representation selection (see section 5.13.3, page 94.)

5.11.4 Generic Arithmetic

In Common Lisp, arithmetic operations are *generic*.[3] The + function can be passed fixnums, bignums, ratios, and various kinds of floats and complexes, in any combination. In addition to the inherent complexity of bignum and ratio operations, there is also a lot of overhead in just figuring out which operation to do and what contagion and canonicalization rules apply. The complexity of generic arithmetic is so great that it is inconceivable to open code it. Instead, the compiler does a function call to a generic arithmetic routine, consuming many instructions before the actual computation even starts.

This is ridiculous, since even Common Lisp programs do a lot of arithmetic, and the hardware is capable of doing operations on small integers and floats with a single instruction. To get acceptable efficiency, the compiler special-cases uses of generic arithmetic that are directly implemented in the hardware. In order to open code arithmetic, several constraints must be met:

- All the arguments must be known to be a good type of number.
- The result must be known to be a good type of number.
- Any intermediate values such as the result of (+ a b) in the call (+ a b c) must be known to be a good type of number.
- All the above numbers with good types must be of the *same* good type. Don't try to mix integers and floats or different float formats.

The "good types" are (signed-byte 32), (unsigned-byte 32), single-float and double-float. See sections 5.11.5, 5.11.6 and 5.11.7 for more discussion of good numeric types.

float is not a good type, since it might mean either single-float or double-float. integer is not a good type, since it might mean bignum. rational is not a good type, since it might mean ratio. Note however that these types are still useful in declarations, since type inference may be able to strengthen a weak declaration into a good one, when it would be at a loss if there was no declaration at all (see section 5.3, page 61). The integer and unsigned-byte (or non-negative integer) types are especially useful in this regard, since they can often be strengthened to a good integer type.

Arithmetic with complex numbers is inefficient in comparison to float and integer arithmetic. Complex numbers are always represented with a pointer descriptor (causing consing overhead), and complex arithmetic is always closed coded using the general generic arithmetic functions. But arithmetic with complex types such as:

```
(complex float)
(complex fixnum)
```

[3] As Steele notes in CLTL II, this is a generic conception of generic, and is not to be confused with the CLOS concept of a generic function.

is still faster than bignum or ratio arithmetic, since the implementation is much simpler.

Note: don't use / to divide integers unless you want the overhead of rational arithmetic. Use truncate even when you know that the arguments divide evenly.

You don't need to remember all the rules for how to get open-coded arithmetic, since efficiency notes will tell you when and where there is a problem—see section 5.13, page 92.

5.11.5 Fixnums

A fixnum is a "FIXed precision NUMber". In modern Common Lisp implementations, fixnums can be represented with an immediate descriptor, so operating on fixnums requires no consing or memory references. Clever choice of representations also allows some arithmetic operations to be done on fixnums using hardware supported word-integer instructions, somewhat reducing the speed penalty for using an unnatural integer representation.

It is useful to distinguish the fixnum type from the fixnum representation of integers. In Python, there is absolutely nothing magical about the fixnum type in comparison to other finite integer types. fixnum is equivalent to (is defined with deftype to be) (signed-byte 30). fixnum is simply the largest subset of integers that *can be represented* using an immediate fixnum descriptor.

Unlike in other Common Lisp compilers, it is in no way desirable to use the fixnum type in declarations in preference to more restrictive integer types such as bit, (integer -43 7) and (unsigned-byte 8). Since Python does understand these integer types, it is preferable to use the more restrictive type, as it allows better type inference (see section 5.3.4, page 62.)

The small, efficient fixnum is contrasted with bignum, or "BIG NUMber". This is another descriptor representation for integers, but this time a pointer representation that allows for arbitrarily large integers. Bignum operations are less efficient than fixnum operations, both because of the consing and memory reference overheads of a pointer descriptor, and also because of the inherent complexity of extended precision arithmetic. While fixnum operations can often be done with a single instruction, bignum operations are so complex that they are always done using generic arithmetic.

A crucial point is that the compiler will use generic arithmetic if it can't *prove* that all the arguments, intermediate values, and results are fixnums. With bounded integer types such as fixnum, the result type proves to be especially problematical, since these types are not closed under common arithmetic operations such as +, -, * and /. For example, (1+ (the fixnum x)) does not necessarily evaluate to a fixnum. Bignums were added to Common Lisp to get around this problem, but they really just transform the correctness problem "if this add overflows, you will get the wrong answer" to the efficiency problem "if this add *might* overflow then your program will run slowly (because of generic arithmetic.)"

There is just no getting around the fact that the hardware only directly supports short integers. To get the most efficient open coding, the compiler must be able to prove that the result is a good integer type. This is an argument in favor of using more restrictive integer types: (1+ (the fixnum x)) may not always be a fixnum, but (1+ (the (unsigned-byte 8) x)) always is. Of course, you can also assert the result type by putting in lots of the declarations and then compiling with safety 0.

5.11.6 Word Integers

Python is unique in its efficient implementation of arithmetic on full-word integers through non-descriptor representations and open coding. Arithmetic on any subtype of these types:

```
(signed-byte 32)
(unsigned-byte 32)
```

is reasonably efficient, although subtypes of fixnum remain somewhat more efficient.

If a word integer must be represented as a descriptor, then the bignum representation is used, with its associated consing overhead. The support for word integers in no way changes the language semantics, it just makes arithmetic on small bignums vastly more efficient. It is fine to do arithmetic operations with mixed fixnum and word integer operands; just declare the most specific integer type you can, and let the compiler decide what representation to use.

In fact, to most users, the greatest advantage of word integer arithmetic is that it effectively provides a few guard bits on the fixnum representation. If there are missing assertions on intermediate values in a fixnum expression, the intermediate results can usually be proved to fit in a word. After the whole expression is evaluated, there will often be a fixnum assertion on the final result, allowing creation of a fixnum result without even checking for overflow.

The remarks in section 5.11.5 about fixnum result type also apply to word integers; you must be careful to give the compiler enough information to prove that the result is still a word integer. This time, though, when we blow out of word integers we land in into generic bignum arithmetic, which is much worse than sleazing from fixnums to word integers. Note that mixing (unsigned-byte 32) arguments with arguments of any signed type (such as fixnum) is a no-no, since the result might not be unsigned.

5.11.7 Floating Point Efficiency

Arithmetic on objects of type single-float and double-float is efficiently implemented using non-descriptor representations and open coding. As for integer arithmetic, the arguments must be known to be of the same float type. Unlike for integer arithmetic, the results and intermediate values usually take care of themselves due to the rules of float contagion, i.e. (1+ (the single-float x)) is always a single-float.

Although they are not specially implemented, short-float and long-float are also acceptable in declarations, since they are synonyms for the single-float and double-float types, respectively.

Some versions of CMUCL include extra support for floating point arithmetic. In particular, if *features* includes :propagate-float-type, list-style float type specifiers such as (single-float 0.0 1.0) will be used to good effect.

For example, in this function,

```
(defun square (x)
  (declare (type (single-float 0f0 10f0)))
  (* x x))
```

Python can deduce that the return type of the function square is (single-float 0f0 100f0).

Many union types are also supported so that

```
(+ (the (or (integer 1 1) (integer 5 5)) x)
   (the (or (integer 10 10) (integer 20 20)) y))
```

has the inferred type (or (integer 11 11) (integer 15 15) (integer 21 21) (integer 25 25)). This also works for floating-point numbers. Member types, however, are not because in general the member elements do not have to be numbers. Thus, instead of (member 1 4), you should write (or (integer 1 1) (integer 4 4)).

In addition, if :propagate-fun-type is in *features*, Python knows how to infer types for many mathematical functions including square root, exponential and logarithmic functions, trignometric functions and their inverses, and hyperbolic functions and their inverses. For numeric code, this can greatly enhance efficiency by allowing the compiler to use specialized versions of the functions instead of the generic versions. The greatest benefit of this type inference is determining that the result of the function is real-valued number instead of possibly being a complex-valued number.

For example, consider the function

```
(defun fun (x)
  (declare (type (single-float 0f0 100f0) x))
  (values (sqrt x) (log x 10f0)))
```

With this declaration, the compiler can determine that the argument to sqrt and log are always non-negative so that the result is always a single-float. In fact, the return type for this function is derived to be (values (single-float 0f0 10f0) (single-float * 2f0)).

If the declaration were reduced to just (declare single-float x), the argument to sqrt and log could be negative. This forces the use of the generic versions of these functions because the result could be a complex number.

Union types are not yet supported for functions.

We note, however, that proper interval arithmetic is not fully implemented in the compiler so the inferred types may be slightly in error due to round-off errors. This round-off error could accumulate to cause the compiler to erroneously deduce the result type and cause code to be removed as being unreachable.[4]Thus, the

[4]This, however, has not actually happened, but it is a possibility.

declarations should only be precise enough for the compiler to deduce that a real-valued argument to a function would produce a real-valued result. The efficiency notes (see section 5.13.3, page 94) from the compiler will guide you on what declarations might be useful.

When a float must be represented as a descriptor, a pointer representation is used, creating consing overhead. For this reason, you should try to avoid situations (such as full call and non-specialized data structures) that force a descriptor representation. See sections 5.11.8, 5.11.9 and 5.11.10.

See section 2.1.3, page 5 for information on the extensions to support IEEE floating point.

5.11.8 Specialized Arrays

Common Lisp supports specialized array element types through the :element-type argument to make-array. When an array has a specialized element type, only elements of that type can be stored in the array. From this restriction comes two major efficiency advantages:

- A specialized array can save space by packing multiple elements into a single word. For example, a base-char array can have 4 elements per word, and a bit array can have 32. This space-efficient representation is possible because it is not necessary to separately indicate the type of each element.
- The elements in a specialized array can be given the same non-descriptor representation as the one used in registers and on the stack, eliminating the need for representation conversions when reading and writing array elements. For objects with pointer descriptor representations (such as floats and word integers) there is also a substantial consing reduction because it is not necessary to allocate a new object every time an array element is modified.

These are the specialized element types currently supported:

```
bit
(unsigned-byte 2)
(unsigned-byte 4)
(unsigned-byte 8)
(unsigned-byte 16)
(unsigned-byte 32)
base-character
single-float
double-float
```

Some versions of CMUCL[5] also support the following specialized element types:

```
(signed-byte 8)
(signed-byte 16)
(signed-byte 30)
(signed-byte 32)
```

Although a simple-vector can hold any type of object, t should still be considered a specialized array type, since arrays with element type t are specialized to hold descriptors.

When using non-descriptor representations, it is particularly important to make sure that array accesses are open-coded, since in addition to the generic operation overhead, efficiency is lost when the array element is converted to a descriptor so that it can be passed to (or from) the generic access routine. You can detect inefficient array accesses by enabling efficiency notes, see section 5.13, page 92. See section 5.10.3, page 83.

5.11.9 Specialized Structure Slots

Structure slots declared by the :type defstruct slot option to have certain known numeric types are also given non-descriptor representations. These types (and subtypes of these types) are supported:

```
(unsigned-byte 32)
single-float
double-float
```

[5]Currently, this includes the x86 and Sparc versions which are compiled with the :signed-array feature.

The primary advantage of specialized slot representations is a large reduction spurious memory allocation and access overhead of programs that intensively use these types.

5.11.10 Interactions With Local Call

Local call has many advantages (see section 5.6, page 72); one relevant to our discussion here is that local call extends the usefulness of non-descriptor representations. If the compiler knows from the argument type that an argument has a non-descriptor representation, then the argument will be passed in that representation. The easiest way to ensure that the argument type is known at compile time is to always declare the argument type in the called function, like:

```
(defun 2+f (x)
  (declare (single-float x))
  (+ x 2.0))
```

The advantages of passing arguments and return values in a non-descriptor representation are the same as for non-descriptor representations in general: reduced consing and memory access (see section 5.11.2, page 85.) This extends the applicative programming styles discussed in section 5.6 to numeric code. Also, if source files are kept reasonably small, block compilation can be used to reduce number consing to a minimum.

Note that non-descriptor return values can only be used with the known return convention (section 5.6.5.) If the compiler can't prove that a function always returns the same number of values, then it must use the unknown values return convention, which requires a descriptor representation. Pay attention to the known return efficiency notes to avoid number consing.

5.11.11 Representation of Characters

Python also uses a non-descriptor representation for characters when convenient. This improves the efficiency of string manipulation, but is otherwise pretty invisible; characters have an immediate descriptor representation, so there is not a great penalty for converting a character to a descriptor. Nonetheless, it may sometimes be helpful to declare character-valued variables as base-character.

5.12 General Efficiency Hints

This section is a summary of various implementation costs and ways to get around them. These hints are relatively unrelated to the use of the Python compiler, and probably also apply to most other Common Lisp implementations. In each section, there are references to related in-depth discussion.

5.12.1 Compile Your Code

At this point, the advantages of compiling code relative to running it interpreted probably need not be emphasized too much, but remember that in CMUCL, compiled code typically runs hundreds of times faster than interpreted code. Also, compiled (fasl) files load significantly faster than source files, so it is worthwhile compiling files which are loaded many times, even if the speed of the functions in the file is unimportant.

Even disregarding the efficiency advantages, compiled code is as good or better than interpreted code. Compiled code can be debugged at the source level (see chapter 3), and compiled code does more error checking. For these reasons, the interpreter should be regarded mainly as an interactive command interpreter, rather than as a programming language implementation.

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

5.12.2 Avoid Unnecessary Consing

Consing is another 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. Arithmetic and function call can also have hidden consing overheads. Consing hurts performance in the following ways:

- Consing reduces 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 by starting a new lisp process.

Consing is not undiluted evil, since programs do things other than consing, and appropriate consing can speed up the real work. It would certainly save time to allocate a vector of intermediate results that are reused hundreds of times. Also, if it is necessary to copy a large data structure many times, it may be more efficient to update the data structure non-destructively; this somewhat increases update overhead, but makes copying trivial.

Note that the remarks in section 5.1.5 about the importance of separating tuning from coding also apply to consing overhead. The majority of consing will be done by a small portion of the program. The consing hot spots are even less predictable than the CPU hot spots, so don't waste time and create bugs by doing unnecessary consing optimization. During initial coding, avoid unnecessary side-effects and cons where it is convenient. If profiling reveals a consing problem, *then* go back and fix the hot spots.

See section 5.11.2, page 85 for a discussion of how to avoid number consing in Python.

5.12.3 Complex Argument Syntax

Common Lisp has very powerful argument passing mechanisms. Unfortunately, two of the most powerful mechanisms, rest arguments and keyword arguments, have a significant performance penalty:

- With keyword arguments, the called function has to parse the supplied keywords by iterating over them and checking them against the desired keywords.
- With rest arguments, the function must cons a list to hold the arguments. If a function is called many times or with many arguments, large amounts of memory will be allocated.

Although rest argument consing is worse than keyword parsing, neither problem is serious unless thousands of calls are made to such a function. The use of keyword arguments is strongly encouraged in functions with many arguments or with interfaces that are likely to be extended, and rest arguments are often natural in user interface functions.

Optional arguments have some efficiency advantage over keyword arguments, but their syntactic clumsiness and lack of extensibility has caused many Common Lisp programmers to abandon use of optionals except in functions that have obviously simple and immutable interfaces (such as subseq), or in functions that are only called in a few places. When defining an interface function to be used by other programmers or users, use of only required and keyword arguments is recommended.

Parsing of defmacro keyword and rest arguments is done at compile time, so a macro can be used to provide a convenient syntax with an efficient implementation. If the macro-expanded form contains no keyword or rest arguments, then it is perfectly acceptable in inner loops.

Keyword argument parsing overhead can also be avoided by use of inline expansion (see section 5.8, page 79) and block compilation (section 5.7.)

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

5.12.4 Mapping and Iteration

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

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

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

- The creation of lists of intermediate results causes much consing (see 5.12.2).
- 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))
```

See sections 5.3.1 and 5.4.1 for a discussion of the interactions of iteration constructs with type inference and variable optimization. Also, section 5.6.4 discusses an applicative style of iteration.

5.12.5 Trace Files and Disassembly

In order to write efficient code, you need to know the relative costs of different operations. The main reason why writing efficient Common Lisp code is difficult is that there are so many operations, and the costs of these operations vary in obscure context-dependent ways. Although efficiency notes point out some problem areas, the only way to ensure generation of the best code is to look at the assembly code output.

The disassemble function is a convenient way to get the assembly code for a function, but it can be very difficult to interpret, since the correspondence with the original source code is weak. A better (but more awkward) option is to use the :trace-file argument to compile-file to generate a trace file.

A trace file is a dump of the compiler's internal representations, including annotated assembly code. Each component in the program gets four pages in the trace file (separated by "*L*"):

- The implicit-continuation (or IR1) representation of the optimized source. This is a dump of the flow graph representation used for "source level" optimizations. As you will quickly notice, it is not really very close to the source. This representation is not very useful to even sophisticated users.
- The Virtual Machine (VM, or IR2) representation of the program. This dump represents the generated code as sequences of "Virtual OPerations" (VOPs.) This representation is intermediate between the source and the assembly code—each VOP corresponds fairly directly to some primitive function or construct, but a given VOP also has a fairly predictable instruction sequence. An operation (such as +) may have multiple implementations with different cost and applicability. The choice of a particular VOP such as +/fixnum or +/single-float represents this choice of implementation. Once you are familiar with it, the VM representation is probably the most useful for determining what implementation has been used.
- An assembly listing, annotated with the VOP responsible for generating the instructions. This listing is useful for figuring out what a VOP does and how it is implemented in a particular context, but its large size makes it more difficult to read.
- A disassembly of the generated code, which has all pseudo-operations expanded out, but is not annotated with VOPs.

Note that trace file generation takes much space and time, since the trace file is tens of times larger than the source file. To avoid huge confusing trace files and much wasted time, it is best to separate the critical program portion into its own file and then generate the trace file from this small file.

5.13 Efficiency Notes

Efficiency notes are messages that warn the user that the compiler has chosen a relatively inefficient implementation for some operation. Usually an efficiency note reflects the compiler's desire for more type information. If the type of the values concerned is known to the programmer, then additional declarations can be used to get a more efficient implementation.

Efficiency notes are controlled by the extensions:inhibit-warnings (see section 4.7.1, page 50) optimization quality. When speed is greater than extensions:inhibit-warnings, efficiency notes are enabled. Note that this implicitly enables efficiency notes whenever speed is increased from its default of 1.

Consider this program with an obscure missing declaration:

```
(defun eff-note (x y z)
  (declare (fixnum x y z))
  (the fixnum (+ x y z)))
```

If compiled with (speed 3) (safety 0), this note is given:

```
In: DEFUN EFF-NOTE
  (+ X Y Z)
==>
  (+ (+ X Y) Z)
Note: Forced to do inline (signed-byte 32) arithmetic (cost 3).
      Unable to do inline fixnum arithmetic (cost 2) because:
      The first argument is a (INTEGER -1073741824 1073741822),
      not a FIXNUM.
```

This efficiency note tells us that the result of the intermediate computation (+ x y) is not known to be a fixnum, so the addition of the intermediate sum to z must be done less efficiently. This can be fixed by changing the definition of eff-note:

```
(defun eff-note (x y z)
  (declare (fixnum x y z))
  (the fixnum (+ (the fixnum (+ x y)) z)))
```

5.13.1 Type Uncertainty

The main cause of inefficiency is the compiler's lack of adequate information about the types of function argument and result values. Many important operations (such as arithmetic) have an inefficient general (generic) case, but have efficient implementations that can usually be used if there is sufficient argument type information.

Type efficiency notes are given when a value's type is uncertain. There is an important distinction between values that are *not known* to be of a good type (uncertain) and values that are *known not* to be of a good type. Efficiency notes are given mainly for the first case (uncertain types.) If it is clear to the compiler that that there is not an efficient implementation for a particular function call, then an efficiency note will only be given if the extensions:inhibit-warnings optimization quality is 0 (see section 4.7.1, page 50.)

In other words, the default efficiency notes only suggest that you add declarations, not that you change the semantics of your program so that an efficient implementation will apply. For example, compilation of this form will not give an efficiency note:

```
(elt (the list l) i)
```

even though a vector access is more efficient than indexing a list.

5.13.2 Efficiency Notes and Type Checking

It is important that the eff-note example above used (safety 0). When type checking is enabled, you may get apparently spurious efficiency notes. With (safety 1), the note has this extra line on the end:

```
The result is a (INTEGER -1610612736 1610612733), not a FIXNUM.
```

This seems strange, since there is a the declaration on the result of that second addition.

In fact, the inefficiency is real, and is a consequence of Python's treating declarations as assertions to be verified. The compiler can't assume that the result type declaration is true—it must generate the result and then test whether it is of the appropriate type.

In practice, this means that when you are tuning a program to run without type checks, you should work from the efficiency notes generated by unsafe compilation. If you want code to run efficiently with type checking, then you should pay attention to all the efficiency notes that you get during safe compilation. Since user supplied output type assertions (e.g., from the) are disregarded when selecting operation implementations for safe code, you must somehow give the compiler information that allows it to prove that the result truly must be of a good type. In our example, it could be done by constraining the argument types more:

```
(defun eff-note (x y z)
  (declare (type (unsigned-byte 18) x y z))
  (+ x y z))
```

Of course, this declaration is acceptable only if the arguments to eff-note always *are* (unsigned-byte 18) integers.

5.13.3 Representation Efficiency Notes

When operating on values that have non-descriptor representations (see section 5.11.2, page 85), there can be a substantial time and consing penalty for converting to and from descriptor representations. For this reason, the compiler gives an efficiency note whenever it is forced to do a representation coercion more expensive than *efficiency-note-cost-threshold* (page 95).

Inefficient representation coercions may be due to type uncertainty, as in this example:

```
(defun set-flo (x)
  (declare (single-float x))
  (prog ((var 0.0))
    (setq var (gorp))
    (setq var x)
    (return var)))
```

which produces this efficiency note:

```
In: DEFUN SET-FLO
  (SETQ VAR X)
Note: Doing float to pointer coercion (cost 13) from X to VAR.
```

The variable var is not known to always hold values of type single-float, so a descriptor representation must be used for its value. In sort of situation, and adding a declaration will eliminate the inefficiency.

Often inefficient representation conversions are not due to type uncertainty—instead, they result from evaluating a non-descriptor expression in a context that requires a descriptor result:

- Assignment to or initialization of any data structure other than a specialized array (see section 5.11.8, page 89), or
- Assignment to a special variable, or
- Passing as an argument or returning as a value in any function call that is not a local call (see section 5.11.10, page 90.)

If such inefficient coercions appear in a "hot spot" in the program, data structures redesign or program reorganization may be necessary to improve efficiency. See sections 5.7, 5.11 and 5.14.

Because representation selection is done rather late in compilation, the source context in these efficiency notes is somewhat vague, making interpretation more difficult. This is a fairly straightforward example:

```
(defun cf+ (x y)
  (declare (single-float x y))
  (cons (+ x y) t))
```

which gives this efficiency note:

```
In: DEFUN CF+
  (CONS (+ X Y) T)
Note: Doing float to pointer coercion (cost 13), for:
      The first argument of CONS.
```

The source context form is almost always the form that receives the value being coerced (as it is in the preceding example), but can also be the source form which generates the coerced value. Compiling this example:

```
(defun if-cf+ (x y)
  (declare (single-float x y))
  (cons (if (grue) (+ x y) (snoc)) t))
```

produces this note:

```
In: DEFUN IF-CF+
  (+ X Y)
Note: Doing float to pointer coercion (cost 13).
```

In either case, the note's text explanation attempts to include additional information about what locations are the source and destination of the coercion. Here are some example notes:

```
  (IF (GRUE) X (SNOC))
Note: Doing float to pointer coercion (cost 13) from X.

  (SETQ VAR X)
Note: Doing float to pointer coercion (cost 13) from X to VAR.
```

Note that the return value of a function is also a place to which coercions may have to be done:

```
  (DEFUN F+ (X Y) (DECLARE (SINGLE-FLOAT X Y)) (+ X Y))
Note: Doing float to pointer coercion (cost 13) to "<return value>".
```

Sometimes the compiler is unable to determine a name for the source or destination, in which case the source context is the only clue.

5.13.4 Verbosity Control

These variables control the verbosity of efficiency notes:

efficiency-note-cost-threshold [*Variable*]

Before printing some efficiency notes, the compiler compares the value of this variable to the difference in cost between the chosen implementation and the best potential implementation. If the difference is not greater than this limit, then no note is printed. The units are implementation dependent; the initial value suppresses notes about "trivial" inefficiencies. A value of 1 will note any inefficiency.

efficiency-note-limit [*Variable*]

When printing some efficiency notes, the compiler reports possible efficient implementations. The initial value of 2 prevents excessively long efficiency notes in the common case where there is no type information, so all implementations are possible.

5.14 Profiling

The first step in improving a program's performance is to profile the activity of the program to find where it spends its time. The best way to do this is to use the profiling utility found in the profile package. This package provides a macro profile that encapsulates functions with statistics gathering code.

5.14.1 Profile Interface

profile:*timed-functions* [*Variable*]

This variable holds a list of all functions that are currently being profiled.

profile:profile {*name* |:callers t}* [*Macro*]

This macro wraps profiling code around the named functions. As in trace, the *name*s are not evaluated. If a function is already profiled, then the function is unprofiled and reprofiled (useful to notice function redefinition.) A warning is printed for each name that is not a defined function.

If :callers *t* is specified, then each function that calls this function is recorded along with the number of calls made.

profile:unprofile {*name*}* [*Macro*]

This macro removes profiling code from the named functions. If no *name*s are supplied, all currently profiled functions are unprofiled.

profile:profile-all &key :package :callers-p [*Macro*]

This macro in effect calls profile:profile for each function in the specified package which defaults to *package*. :callers-p has the same meaning as in profile:profile.

profile:report-time {*name*}* [*Macro*]

This macro prints a report for each *name*d function of the following information:

- The total CPU time used in that function for all calls,
- the total number of bytes consed in that function for all calls,
- the total number of calls,
- the average amount of CPU time per call.

Summary totals of the CPU time, consing and calls columns are printed. An estimate of the profiling overhead is also printed (see below). If no *name*s are supplied, then the times for all currently profiled functions are printed.

reset-time {*name*}* [*Macro*]

This macro resets the profiling counters associated with the *name*d functions. If no *name*s are supplied, then all currently profiled functions are reset.

5.14.2 Profiling Techniques

Start by profiling big pieces of a program, then carefully choose which functions close to, but not in, the inner loop are to be profiled next. Avoid profiling functions that are called by other profiled functions, since this opens the possibility of profiling overhead being included in the reported times.

If the per-call time reported is less than 1/10 second, then consider the clock resolution and profiling overhead before you believe the time. It may be that you will need to run your program many times in order to average out to a higher resolution.

5.14.3 Nested or Recursive Calls

The profiler attempts to compensate for nested or recursive calls. Time and consing overhead will be charged to the dynamically innermost (most recent) call to a profiled function. So profiling a subfunction of a profiled function will cause the reported time for the outer function to decrease. However if an inner function has a large number of calls, some of the profiling overhead may "leak" into the reported time for the outer function. In general, be wary of profiling short functions that are called many times.

5.14.4 Clock resolution

Unless you are very lucky, the length of your machine's clock "tick" is probably much longer than the time it takes simple function to run. For example, on the IBM RT, the clock resolution is 1/50 second. This means that if a function is only called a few times, then only the first couple decimal places are really meaningful.

Note however, that if a function is called many times, then the statistical averaging across all calls should result in increased resolution. For example, on the IBM RT, if a function is called a thousand times, then a resolution of tens of microseconds can be expected.

5.14.5 Profiling overhead

The added profiling code takes time to run every time that the profiled function is called, which can disrupt the attempt to collect timing information. In order to avoid serious inflation of the times for functions that take little time to run, an estimate of the overhead due to profiling is subtracted from the times reported for each function.

Although this correction works fairly well, it is not totally accurate, resulting in times that become increasingly meaningless for functions with short runtimes. This is only a concern when the estimated profiling overhead is many times larger than reported total CPU time.

The estimated profiling overhead is not represented in the reported total CPU time. The sum of total CPU time and the estimated profiling overhead should be close to the total CPU time for the entire profiling run (as determined by the time macro.) Time unaccounted for is probably being used by functions that you forgot to profile.

5.14.6 Additional Timing Utilities

time *form* [*Macro*]

This macro evaluates *form*, prints some timing and memory allocation information to *trace-output*, and returns any values that *form* returns. The timing information includes real time, user run time, and system run time. This macro executes a form and reports the time and consing overhead. If the time form is not compiled (e.g. it was typed at top-level), then compile will be called on the form to give more accurate timing information. If you really want to time interpreted speed, you can say:

```
(time (eval 'form))
```

Things that execute fairly quickly should be timed more than once, since there may be more paging overhead in the first timing. To increase the accuracy of very short times, you can time multiple evaluations:

```
(time (dotimes (i 100) form))
```

extensions:get-bytes-consed [*Function*]

This function returns the number of bytes allocated since the first time you called it. The first time it is called it returns zero. The above profiling routines use this to report consing information.

extensions:*gc-run-time* [*Variable*]

This variable accumulates the run-time consumed by garbage collection, in the units returned by get-internal-run-time.

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

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

5.14.7 A Note on Timing

There are two general kinds of timing information provided by the `time` macro and other profiling utilities: real time and run time. Real time is elapsed, wall clock time. It will be affected in a fairly obvious way by any other activity on the machine. The more other processes contending for CPU and memory, the more real time will increase. This means that real time measurements are difficult to replicate, though this is less true on a dedicated workstation. The advantage of real time is that it is real. It tells you really how long the program took to run under the benchmarking conditions. The problem is that you don't know exactly what those conditions were.

Run time is the amount of time that the processor supposedly spent running the program, as opposed to waiting for I/O or running other processes. "User run time" and "system run time" are numbers reported by the Unix kernel. They are supposed to be a measure of how much time the processor spent running your "user" program (which will include GC overhead, etc.), and the amount of time that the kernel spent running "on your behalf."

Ideally, user time should be totally unaffected by benchmarking conditions; in reality user time does depend on other system activity, though in rather non-obvious ways.

System time will clearly depend on benchmarking conditions. In Lisp benchmarking, paging activity increases system run time (but not by as much as it increases real time, since the kernel spends some time waiting for the disk, and this is not run time, kernel or otherwise.)

In my experience, the biggest trap in interpreting kernel/user run time is to look only at user time. In reality, it seems that the *sum* of kernel and user time is more reproducible. The problem is that as system activity increases, there is a spurious *decrease* in user run time. In effect, as paging, etc., increases, user time leaks into system time.

So, in practice, the only way to get truly reproducible results is to run with the same competing activity on the system. Try to run on a machine with nobody else logged in, and check with "ps aux" to see if there are any system processes munching large amounts of CPU or memory. If the ratio between real time and the sum of user and system time varies much between runs, then you have a problem.

5.14.8 Benchmarking Techniques

Given these imperfect timing tools, how do should you do benchmarking? The answer depends on whether you are trying to measure improvements in the performance of a single program on the same hardware, or if you are trying to compare the performance of different programs and/or different hardware.

For the first use (measuring the effect of program modifications with constant hardware), you should look at *both* system+user and real time to understand what effect the change had on CPU use, and on I/O (including paging.) If you are working on a CPU intensive program, the change in system+user time will give you a moderately reproducible measure of performance across a fairly wide range of system conditions. For a CPU intensive program, you can think of system+user as "how long it would have taken to run if I had my own machine." So in the case of comparing CPU intensive programs, system+user time is relatively real, and reasonable to use.

For programs that spend a substantial amount of their time paging, you really can't predict elapsed time under a given operating condition without benchmarking in that condition. User or system+user time may be fairly reproducible, but it is also relatively meaningless, since in a paging or I/O intensive program, the program is spending its time waiting, not running, and system time and user time are both measures of run time. A change that reduces run time might increase real time by increasing paging.

Another common use for benchmarking is comparing the performance of the same program on different hardware. You want to know which machine to run your program on. For comparing different machines (operating systems, etc.), the only way to compare that makes sense is to set up the machines in *exactly* the way that they will *normally* be run, and then measure *real* time. If the program will normally be run along with X, then run X. If the program will normally be run on a dedicated workstation, then be sure nobody else is on the benchmarking machine. If the program will normally be run on a machine with three other Lisp jobs, then run three other Lisp jobs. If the program will normally be run on a machine with 64MB of memory, then run with 64MB. Here, "normal" means "normal for that machine".

If you have a program you believe to be CPU intensive, then you might be tempted to compare "run" times across systems, hoping to get a meaningful result even if the benchmarking isn't done under the expected running condition. Don't to this, for two reasons:

- The operating systems might not compute run time in the same way.
- Under the real running condition, the program might not be CPU intensive after all.

In the end, only real time means anything—it is the amount of time you have to wait for the result. The only valid uses for run time are:

- To develop insight into the program. For example, if run time is much less than elapsed time, then you are probably spending lots of time paging.
- To evaluate the relative performance of CPU intensive programs in the same environment.

Chapter 6

UNIX Interface

by Robert MacLachlan, Skef Wholey,

Bill Chiles and William Lott

CMUCL attempts to make the full power of the underlying environment available to the Lisp programmer. This is done using combination of hand-coded interfaces 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 Unix and X11, as well as access to the standard system documentation.

6.1 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.

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

Returns the first non-null value from cmd-switch-value, the first element in cmd-switch-words, or the first word in *command-line-words*.

extensions:get-command-line-switch *sname* [*Function*]

This function takes the name of a switch as a string and returns the value of the switch given on the command line. If no value was specified, then any following words are returned. If there are no following words, then t is returned. If the switch was not specified, then nil is returned.

extensions:defswitch *name* &optional *function* [*Macro*]

This macro causes *function* to be called when the switch *name* appears in the command line. Name is a simple-string that does not begin with a hyphen (unless the switch name really does begin with one.)

If *function* is not supplied, then the switch is parsed into *command-line-switches*, but otherwise ignored. This suppresses the undefined switch warning which would otherwise take place. The warning can also be globally suppressed by *complain-about-illegal-switches*.

6.2 Useful Variables

system:*stdin* [*Variable*]
system:*stdout* [*Variable*]
system:*stderr* [*Variable*]

Streams connected to the standard input, output and error file descriptors.

system:*tty* [*Variable*]

A stream connected to '/dev/tty'.

extensions:*environment-list* [*Variable*]

The environment variables inherited by the current process, as a keyword-indexed alist. For example, to access the DISPLAY environment variable, you could use

```
(cdr (assoc :display ext:*environment-list*))
```

Note that the case of the variable name is lost in the conversion to a keyword.

6.3 Lisp Equivalents for C Routines

The UNIX 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>))
```

If there are multiple out or in-out arguments, then there are multiple additional returns values.

Fortunately, CMUCL 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. 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.

6.4 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. Records, arrays, and pointer types are implemented with the Alien facility (see section 8, page 115). 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).

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 memory was obtained from make-alien (page 119) or from a foreign function call to a routine that used malloc, then free-alien (page 119) should be used.

6.5 System Area Pointers

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 Area Pointers (SAPs) provide a mechanism that bypasses the Alien type system and accesses virtual memory directly. A SAP is a raw byte pointer into the lisp process address space. SAPs are represented with a pointer descriptor, so SAP creation can cause consing. However, the compiler uses a non-descriptor representation for SAPs when possible, so the consing overhead is generally minimal. See section 5.11.2, page 85.

system:sap-int *sap* [*Function*]
system:int-sap *int* [*Function*]

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. The integer representation of a SAP is the byte offset of the SAP from the start of the address space.

system:sap+ *sap offset* [*Function*]

This function adds a byte *offset* to *sap*, returning a new SAP.

system:sap-ref-8 *sap offset* [*Function*]
system:sap-ref-16 *sap offset* [*Function*]
system:sap-ref-32 *sap offset* [*Function*]

These functions return the 8, 16 or 32 bit unsigned integer at *offset* from *sap*. The *offset* is always a byte offset, regardless of the number of bits accessed. setf may be used with the these functions to deposit values into virtual memory.

system:signed-sap-ref-8 *sap offset* [*Function*]
system:signed-sap-ref-16 *sap offset* [*Function*]
system:signed-sap-ref-32 *sap offset* [*Function*]

These functions are the same as the above unsigned operations, except that they sign-extend, returning a negative number if the high bit is set.

6.6 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 Unix package. The name of the interface for a particular system call is the name of the system call prepended with unix-. The system usually defines the associated constants without any prefix name. To find out how to use a particular system call, try using describe on it. If that is unhelpful, look at the source in 'unix.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-nil, often t.

For example, to use the chdir syscall:

```
(multiple-value-bind (success errno)
    (unix:unix-chdir "/tmp")
  (unless success
     (error "Can't change working directory: ~a"
            (unix:get-unix-error-msg errno))))
```

Unix:get-unix-error-msg *error* [*Function*]

This function returns a string describing the Unix error number *error* (this is similar to the Unix function perror).

6.7 File Descriptor Streams

Many of the UNIX system calls return file descriptors. Instead of using other UNIX system calls to perform I/O on them, you can create a stream around them. For this purpose, fd-streams exist. See also read-n-bytes (page 15).

system:make-fd-stream *descriptor* &key :input :output :element-type :buffering :name :file :original :delete-original :auto-close :timeout :pathname [*Function*]

This function creates a file descriptor stream using *descriptor*. If :input is non-nil, input operations are allowed. If :output is non-nil, output operations are allowed. The default is input only. These keywords are defined:

:element-type
: is the type of the unit of transaction for the stream, which defaults to string-char. See the Common Lisp description of open for valid values.

:buffering
: is the kind of output buffering desired for the stream. Legal values are :none for no buffering, :line for buffering up to each newline, and :full for full buffering.

:name
: is a simple-string name to use for descriptive purposes when the system prints an fd-stream. When printing fd-streams, the system prepends the streams name with Stream for . If *name* is unspecified, it defaults to a string containing *file* or *descriptor*, in order of preference.

:file, :original
: *file* specifies the defaulted namestring of the associated file when creating a file stream (must be a simple-string). *original* is the simple-string name of a backup file containing the original contents of *file* while writing *file*.

When you abort the stream by passing t to close as the second argument, if you supplied both *file* and *original*, close will rename the *original* name to the *file* name. When you close the stream normally, if you supplied *original*, and *delete-original* is non-nil, close deletes *original*. If *auto-close* is true (the default), then *descriptor* will be closed when the stream is garbage collected.

:pathname
: : The original pathname passed to open and returned by pathname; not defaulted or translated.

:timeout
: if non-null, then *timeout* is an integer number of seconds after which an input wait should time out. If a read does time out, then the system:io-timeout condition is signalled.

system:fd-stream-p *object* [*Function*]

This function returns t if *object* is an fd-stream, and nil if not. Obsolete: use the portable (typep x 'file-stream).

system:fd-stream-fd *stream* [*Function*]

This returns the file descriptor associated with *stream*.

6.8 Unix Signals

CMUCL allows access to all the Unix signals that can be generated under Unix. 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 Unix package.

6.8.1 Changing Signal Handlers

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

This macro should be called with a list of signal specifications, *specs*. Each element of *specs* should be a list of two 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 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. This macro 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 executing an interrupt handler, the system disables interrupts, as if the handler was wrapped in in a 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 might need to be interrupted.

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* [*Function*]

This function establishes *function* as the handler for *signal*. 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.

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.

6.8.2 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
   (Unix:unix-kill (Unix:unix-getpid) Unix: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 ((Unix:SIGINT #'ih-sigint)
                          (Unix:SIGTSTP #'ih-sigtstp))
  <user code to execute with the above signal handlers enabled.>
)
```

Chapter 7

Event Dispatching with SERVE-EVENT

by Bill Chiles and Robert 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. CMUCL 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. These events include X events and activity on Unix file descriptors. The object-oriented mechanism is only available with the first two, and it is optional with X events as described later in this chapter. In an X event, the window ID is the object capability and the X event type is the operation code. The Unix file descriptor input mechanism simply consists of an association list of a handler to call when input shows up on a particular file descriptor.

7.1 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*]

This function makes a new object set. *Name* is a string used only for purposes of identifying the object set when it is printed. *Default-handler* is the function used as a handler when an undefined operation occurs on an object in the set. You can define operations with the serve-*operation* functions exported the extensions package for X events (see section 7.4, page 109). Objects are added with system:add-xwindow-object. Initially the object set has no objects and no defined operations.

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

This function returns the handler function that is the implementation of the operation corresponding to *operation-code* in *object-set*. When set with setf, the setter function establishes the new handler. The serve-*operation* functions exported from the extensions package for X events (see section 7.4, page 109) call this on behalf of the user when announcing a new operation for an object set.

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

These functions add *port* or *window* to *object-set*. *Object* is an arbitrary Lisp object that is associated with the *port* or *window* capability. *Window* is a CLX window. When an event occurs, system:serve-event passes *object* as an argument to the handler function.

7.2 The SERVE-EVENT Function

The system:serve-event function is the standard way for an application to wait for something to happen. For example, the Lisp system calls system:serve-event when it wants input from X or a terminal stream. The idea behind system:serve-event is that it knows the appropriate action to take when any interesting event happens. If an application calls system:serve-event 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:serve-event &optional *timeout* [*Function*]

This function waits for an event to happen and then dispatches 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 causes system:serve-event to poll for any events immediately available for processing. system:serve-event returns t if it serviced at least one event, and nil otherwise. Depending on the application, when system:serve-event returns t, you might want to call it repeatedly with a timeout of zero until it returns nil.

If input is available on any designated file descriptor, then this calls the appropriate handler function supplied by system:add-fd-handler.

Since events for many different applications may arrive simultaneously, an application waiting for a specific event must loop on system:serve-event until the desired event happens. Since programs such as Hemlock call system:serve-event for input, applications usually do not need to call system:serve-event at all; Hemlock allows other application's handlers to run when it goes into an input wait.

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

This function is similar to system:serve-event, except it serves all the pending events rather than just one. It returns t if it serviced at least one event, and nil otherwise.

7.3 Using SERVE-EVENT with Unix File Descriptors

Object sets are not available for use with file descriptors, as there are only two operations possible on file descriptors: input and output. Instead, a handler for either input or output can be registered with system:serve-event for a specific file descriptor. Whenever any input shows up, or output is possible on this file descriptor, the function associated with the handler for that descriptor is funcalled with the descriptor as it's single argument.

system:add-fd-handler *fd direction function* [*Function*]

This function installs and returns a new handler for the file descriptor *fd*. *direction* can be either :input if the system should invoke the handler when input is available or :output if the system should invoke the handler when output is possible. This returns a unique object representing the handler, and this is a suitable argument for system:remove-fd-handler *function* must take one argument, the file descriptor.

system:remove-fd-handler *handler* [*Function*]

This function removes *handler*, that add-fd-handler must have previously returned.

system:with-fd-handler (*direction fd function*) {*form*}* [*Macro*]

This macro executes the supplied forms with a handler installed using *fd*, *direction*, and *function*. See system:add-fd-handler.

system:wait-until-fd-usable *direction fd* &optional *timeout* [*Function*]

This function waits for up to *timeout* seconds for *fd* to become usable for *direction* (either :input or :output). If *timeout* is nil or unspecified, this waits forever.

system:invalidate-descriptor *fd* [*Function*]

This function removes all handlers associated with *fd*. This should only be used in drastic cases (such as I/O errors, but not necessarily EOF). Normally, you should use remove-fd-handler to remove the specific handler.

7.4 Using SERVE-EVENT with the CLX Interface to X

Remember from section 7.1, an object set is a collection of objects, CLX windows in this case, with some set of operations, event keywords, with corresponding implementations, the same handler functions. Since X allows multiple display connections from a given process, you can avoid using object sets if every window in an application or display connection behaves the same. If a particular X application on a single display connection has windows that want to handle certain events differently, then using object sets is a convenient way to organize this since you need some way to map the window/event combination to the appropriate functionality.

The following is a discussion of functions exported from the extensions package that facilitate handling CLX events through system:serve-event. The first two routines are useful regardless of whether you use system:serve-event:

ext:open-clx-display &optional *string* [*Function*]

This function parses *string* for an X display specification including display and screen numbers. *String* defaults to the following:

```
(cdr (assoc :display ext:*environment-list* :test #'eq))
```

If any field in the display specification is missing, this signals an error. ext:open-clx-display returns the CLX display and screen.

ext:flush-display-events *display* [*Function*]

This function flushes all the events in *display*'s event queue including the current event, in case the user calls this from within an event handler.

7.4.1 Without Object Sets

Since most applications that use CLX, can avoid the complexity of object sets, these routines are described in a separate section. The routines described in the next section that use the object set mechanism are based on these interfaces.

ext:enable-clx-event-handling *display handler* [*Function*]

This function causes system:serve-event to notice when there is input on *display*'s connection to the X11 server. When this happens, system:serve-event invokes *handler* on *display* in a dynamic context with an error handler bound that flushes all events from *display* and returns. By returning, the error handler declines to handle the error, but it will have cleared all events; thus, entering the debugger will not result in infinite errors due to streams that wait via system:serve-event for input. Calling this repeatedly on the same *display* establishes *handler* as a new handler, replacing any previous one for *display*.

ext:disable-clx-event-handling *display* [*Function*]

This function undoes the effect of ext:enable-clx-event-handling.

ext:with-clx-event-handling (*display handler*) {*form*}* [*Macro*]

This macro evaluates each *form* in a context where system:serve-event invokes *handler* on *display* whenever there is input on *display*'s connection to the X server. This destroys any previously established handler for *display*.

7.4.2 With Object Sets

This section discusses the use of object sets and system:serve-event to handle CLX events. This is necessary when a single X application has distinct windows that want to handle the same events in different ways. Basically, you need some way of asking for a given window which way you want to handle some event because this event is handled differently depending on the window. Object sets provide this feature.

For each CLX event-key symbol-name ıXXX (for example, *key-press*), there is a function serve-ıXXX of two arguments, an object set and a function. The serve-ıXXX function establishes the function as the handler for the :XXX event in the object set. Recall from section 7.1, system:add-xwindow-object associates some Lisp object with a CLX window in an object set. When system:serve-event notices activity on a window, it calls the function given to ext:enable-clx-event-handling. If this function is ext:object-set-event-handler, it calls the function given to serve-ıXXX, passing the object given to system:add-xwindow-object and the event's slots as well as a couple other arguments described below.

To use object sets in this way:

- Create an object set.
- Define some operations on it using the serve-ıXXX functions.
- Add an object for every window on which you receive requests. This can be the CLX window itself or some structure more meaningful to your application.
- Call system:serve-event to service an X event.

ext:object-set-event-handler *display* [*Function*]

This function is a suitable argument to ext:enable-clx-event-handling. The actual event handlers defined for particular events within a given object set must take an argument for every slot in the appropriate event. In addition to the event slots, ext:object-set-event-handler passes the following arguments:

- The object, as established by system:add-xwindow-object, on which the event occurred.
- event-key, see xlib:event-case.
- send-event-p, see xlib:event-case.

Describing any ext:serve-*event-key-name* function, where *event-key-name* is an event-key symbol-name (for example, ext:serve-key-press), indicates exactly what all the arguments are in their correct order.

When creating an object set for use with ext:object-set-event-handler, specify ext:default-clx-event-handler as the default handler for events in that object set. If no default handler is specified, and the system invokes the default default handler, it will cause an error since this function takes arguments suitable for handling port messages.

7.5 A SERVE-EVENT Example

This section contains two examples using system:serve-event. The first one does not use object sets, and the second, slightly more complicated one does.

7.5.1 Without Object Sets Example

This example defines an input handler for a CLX display connection. It only recognizes :key-press events. The body of the example loops over system:serve-event to get input.

```
(in-package "SERVER-EXAMPLE")

(defun my-input-handler (display)
  (xlib:event-case (display :timeout 0)
    (:key-press (event-window code state)
     (format t "KEY-PRESSED (Window = ~D) = ~S.~%"
```

```
                 (xlib:window-id event-window)
            ;; See Hemlock Command Implementor's Manual for convenient
            ;; input mapping function.
            (ext:translate-character display code state))
      ;; Make XLIB:EVENT-CASE discard the event.
      t)))

(defun server-example ()
  "An example of using the SYSTEM:SERVE-EVENT function and object sets to
   handle CLX events."
  (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)
                                :x 0 :y 0 :width 200 :height 200
                                :background white :border black
                                :border-width 2
                                :event-mask
                                (xlib:make-event-mask :key-press))))
    ;; Wrap code in UNWIND-PROTECT, so we clean up after ourselves.
    (unwind-protect
        (progn
          ;; Enable event handling on the display.
          (ext:enable-clx-event-handling display #'my-input-handler)
          ;; Map the windows to the screen.
          (map-window window)
          ;; Make sure we send all our requests.
          (display-force-output display)
          ;; Call serve-event for 100,000 events or immediate timeouts.
          (dotimes (i 100000) (system:serve-event)))
      ;; Disable event handling on this display.
      (ext:disable-clx-event-handling display)
      ;; Get rid of the window.
      (destroy-window window)
      ;; Pick off any events the X server has already queued for our
      ;; windows, so we don't choke since SYSTEM:SERVE-EVENT is no longer
      ;; prepared to handle events for us.
      (loop
       (unless (deleting-window-drop-event *display* window)
        (return)))
      ;; Close the display.
      (xlib:close-display display))))

(defun deleting-window-drop-event (display win)
  "Check for any events on win.  If there is one, remove it from the
   event queue and return t; otherwise, return nil."
  (xlib:display-finish-output display)
  (let ((result nil))
    (xlib:process-event
     display :timeout 0
     :handler #'(lambda (&key event-window &allow-other-keys)
                  (if (eq event-window win)
                      (setf result t)
                      nil)))
    result))
```

7.5.2 With Object Sets Example

This example involves more work, but you get a little more for your effort. It defines two objects, input-box and slider, and establishes a :key-press handler for each object, key-pressed and slider-pressed. We have two object sets because we handle events on the windows manifesting these objects differently, but the events come over the same display connection.

```
(in-package "SERVER-EXAMPLE")

(defstruct (input-box (:print-function print-input-box)
                      (:constructor make-input-box (display window)))
  "Our program knows about input-boxes, and it doesn't care how they
   are implemented."
  display        ; The CLX display on which my input-box is displayed.
  window)        ; The CLX window in which the user types.
;;;
(defun print-input-box (object stream n)
  (declare (ignore n))
  (format stream "#<Input-Box ~S>" (input-box-display object)))

(defvar *input-box-windows*
        (system:make-object-set "Input Box Windows"
                                #'ext:default-clx-event-handler))

(defun key-pressed (input-box event-key event-window root child
                    same-screen-p x y root-x root-y modifiers time
                    key-code send-event-p)
  "This is our :key-press event handler."
  (declare (ignore event-key root child same-screen-p x y
                   root-x root-y time send-event-p))
  (format t "KEY-PRESSED (Window = ~D) = ~S.~%"
          (xlib:window-id event-window)
          ;; See Hemlock Command Implementor's Manual for convenient
          ;; input mapping function.
          (ext:translate-character (input-box-display input-box)
                                   key-code modifiers)))
;;;
(ext:serve-key-press *input-box-windows* #'key-pressed)

(defstruct (slider (:print-function print-slider)
                   (:include input-box)
                   (:constructor %make-slider
                                 (display window window-width max)))
  "Our program knows about sliders too, and these provide input values
   zero to max."
  bits-per-value  ; bits per discrete value up to max.
  max)            ; End value for slider.
;;;
(defun print-slider (object stream n)
  (declare (ignore n))
  (format stream "#<Slider ~S  0..~D>"
          (input-box-display object)
          (1- (slider-max object))))
;;;
(defun make-slider (display window max)
  (%make-slider display window
                (truncate (xlib:drawable-width window) max)
```

```
                max))

(defvar *slider-windows*
        (system:make-object-set "Slider Windows"
                                #'ext:default-clx-event-handler))

(defun slider-pressed (slider event-key event-window root child
                       same-screen-p x y root-x root-y modifiers time
                       key-code send-event-p)
  "This is our :key-press event handler for sliders.  Probably this is
   a mouse thing, but for simplicity here we take a character typed."
  (declare (ignore event-key root child same-screen-p x y
                   root-x root-y time send-event-p))
  (format t "KEY-PRESSED (Window = ~D) = ~S  -->  ~D.~%"
          (xlib:window-id event-window)
          ;; See Hemlock Command Implementor's Manual for convenient
          ;; input mapping function.
          (ext:translate-character (input-box-display slider)
                                   key-code modifiers)
          (truncate x (slider-bits-per-value slider))))
;;;
(ext:serve-key-press *slider-windows* #'slider-pressed)

(defun server-example ()
  "An example of using the SYSTEM:SERVE-EVENT function and object sets to
   handle CLX events."
  (let* ((display (ext:open-clx-display))
         (screen (display-default-screen display))
         (black (screen-black-pixel screen))
         (white (screen-white-pixel screen))
         (iwindow (create-window :parent (screen-root screen)
                                 :x 0 :y 0 :width 200 :height 200
                                 :background white :border black
                                 :border-width 2
                                 :event-mask
                                 (xlib:make-event-mask :key-press)))
         (swindow (create-window :parent (screen-root screen)
                                 :x 0 :y 300 :width 200 :height 50
                                 :background white :border black
                                 :border-width 2
                                 :event-mask
                                 (xlib:make-event-mask :key-press)))
         (input-box (make-input-box display iwindow))
         (slider (make-slider display swindow 15)))
    ;; Wrap code in UNWIND-PROTECT, so we clean up after ourselves.
    (unwind-protect
        (progn
          ;; Enable event handling on the display.
          (ext:enable-clx-event-handling display
                                         #'ext:object-set-event-handler)
          ;; Add the windows to the appropriate object sets.
          (system:add-xwindow-object iwindow input-box
                                     *input-box-windows*)
          (system:add-xwindow-object swindow slider
                                     *slider-windows*)
          ;; Map the windows to the screen.
```

```
          (map-window iwindow)
          (map-window swindow)
          ;; Make sure we send all our requests.
          (display-force-output display)
          ;; Call server for 100,000 events or immediate timeouts.
          (dotimes (i 100000) (system:serve-event)))
      ;; Disable event handling on this display.
      (ext:disable-clx-event-handling display)
      (delete-window iwindow display)
      (delete-window swindow display)
      ;; Close the display.
      (xlib:close-display display))))

(defun delete-window (window display)
  ;; Remove the windows from the object sets before destroying them.
  (system:remove-xwindow-object window)
  ;; Destroy the window.
  (destroy-window window)
  ;; Pick off any events the X server has already queued for our
  ;; windows, so we don't choke since SYSTEM:SERVE-EVENT is no longer
  ;; prepared to handle events for us.
  (loop
   (unless (deleting-window-drop-event display window)
     (return))))

(defun deleting-window-drop-event (display win)
  "Check for any events on win.  If there is one, remove it from the
   event queue and return t; otherwise, return nil."
  (xlib:display-finish-output display)
  (let ((result nil))
    (xlib:process-event
     display :timeout 0
     :handler #'(lambda (&key event-window &allow-other-keys)
                  (if (eq event-window win)
                      (setf result t)
                      nil)))
    result))
```

Chapter 8

Alien Objects

by Robert MacLachlan and William Lott

8.1 Introduction to Aliens

Because of Lisp's emphasis on dynamic memory allocation and garbage collection, Lisp implementations use unconventional memory representations for objects. This representation mismatch creates problems when a Lisp program must share objects with programs written in another language. There are three different approaches to establishing communication:

- The burden can be placed on the foreign program (and programmer) by requiring the use of Lisp object representations. The main difficulty with this approach is that either the foreign program must be written with Lisp interaction in mind, or a substantial amount of foreign "glue" code must be written to perform the translation.
- The Lisp system can automatically convert objects back and forth between the Lisp and foreign representations. This is convenient, but translation becomes prohibitively slow when large or complex data structures must be shared.
- The Lisp program can directly manipulate foreign objects through the use of extensions to the Lisp language. Most Lisp systems make use of this approach, but the language for describing types and expressing accesses is often not powerful enough for complex objects to be easily manipulated.

CMUCL relies primarily on the automatic conversion and direct manipulation approaches: Aliens of simple scalar types are automatically converted, while complex types are directly manipulated in their foreign representation. Any foreign objects that can't automatically be converted into Lisp values are represented by objects of type alien-value. Since Lisp is a dynamically typed language, even foreign objects must have a run-time type; this type information is provided by encapsulating the raw pointer to the foreign data within an alien-value object.

The Alien type language and operations are most similar to those of the C language, but Aliens can also be used when communicating with most other languages that can be linked with C.

8.2 Alien Types

Alien types have a description language based on nested list structure. For example:

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

has the corresponding Alien type:

```
(struct foo
  (a int)
  (b (array (* (struct foo)) 100)))
```

8.2.1 Defining Alien Types

Types may be either named or anonymous. With structure and union types, the name is part of the type specifier, allowing recursively defined types such as:

```
(struct foo (a (* (struct foo))))
```

An anonymous structure or union type is specified by using the name nil. The with-alien (page 119) macro defines a local scope which "captures" any named type definitions. Other types are not inherently named, but can be given named abbreviations using def-alien-type.

alien:def-alien-type name type [*Macro*]

This macro globally defines *name* as a shorthand for the Alien type *type*. When introducing global structure and union type definitions, *name* may be nil, in which case the name to define is taken from the type's name.

8.2.2 Alien Types and Lisp Types

The Alien types form a subsystem of the CMUCL type system. An alien type specifier provides a way to use any Alien type as a Lisp type specifier. For example

```
(typep foo '(alien (* int)))
```

can be used to determine whether foo is a pointer to an int. alien type specifiers can be used in the same ways as ordinary type specifiers (like string.) Alien type declarations are subject to the same precise type checking as any other declaration (section See section 4.5.2, page 47.)

Note that the Alien type system overlaps with normal Lisp type specifiers in some cases. For example, the type specifier (alien single-float) is identical to single-float, since Alien floats are automatically converted to Lisp floats. When type-of is called on an Alien value that is not automatically converted to a Lisp value, then it will return an alien type specifier.

8.2.3 Alien Type Specifiers

Some Alien type names are Common Lispsymbols, but the names are still exported from the alien package, so it is legal to say alien:single-float. These are the basic Alien type specifiers:

* *type* [*Alien type*]

A pointer to an object of the specified *type*. If *type* is t, then it means a pointer to anything, similar to "void *" in ANSI C. Currently, the only way to detect a null pointer is:

```
(zerop (sap-int (alien-sap ptr)))
```

See section 6.5, page 102

array *type* {*dimension*}* [*Alien type*]

An array of the specified *dimensions*, holding elements of type *type*. Note that (* int) and (array int) are considered to be different types when type checking is done; pointer and array types must be explicitly coerced using cast.

Arrays are accessed using deref, passing the indices as additional arguments. Elements are stored in column-major order (as in C), so the first dimension determines only the size of the memory block, and not the layout of the higher dimensions. An array whose first dimension is variable may be specified by using nil as the first dimension. Fixed-size arrays can be allocated as array elements, structure slots or with-alien variables. Dynamic arrays can only be allocated using make-alien (page 119).

struct *name* {*(field type* {*bits*} *)*}* [*Alien type*]

A structure type with the specified *name* and *fields*. Fields are allocated at the same positions used by the implementation's C compiler. *bits* is intended for C-like bit field support, but is currently unused. If *name* is nil, then the type is anonymous.

If a named Alien struct specifier is passed to def-alien-type (page 116) or with-alien (page 119), then this defines, respectively, a new global or local Alien structure type. If no *fields* are specified, then the fields are taken from the current (local or global) Alien structure type definition of *name*.

union *name* {*(field type* {*bits*} *)*}* [*Alien type*]

Similar to struct, but defines a union type. All fields are allocated at the same offset, and the size of the union is the size of the largest field. The programmer must determine which field is active from context.

enum *name* {*spec*}* [*Alien type*]

An enumeration type that maps between integer values and keywords. If *name* is nil, then the type is anonymous. Each *spec* is either a keyword, or a list (*keyword value*). If *integer* is not supplied, then it defaults to one greater than the value for the preceding spec (or to zero if it is the first spec.)

signed {*bits*} [*Alien type*]

A signed integer with the specified number of bits precision. The upper limit on integer precision is determined by the machine's word size. If no size is specified, the maximum size will be used.

integer {*bits*} [*Alien type*]

Identical to signed—the distinction between signed and integer is purely stylistic.

unsigned {*bits*} [*Alien type*]

Like signed, but specifies an unsigned integer.

boolean {*bits*} [*Alien type*]

Similar to an enumeration type that maps 0 to nil and all other values to t. *bits* determines the amount of storage allocated to hold the truth value.

single-float [*Alien type*]

A floating-point number in IEEE single format.

double-float [*Alien type*]

A floating-point number in IEEE double format.

function *result-type* {*arg-type*}* [*Alien type*]

A Alien function that takes arguments of the specified *arg-types* and returns a result of type *result-type*. Note that the only context where a function type is directly specified is in the argument to alien-funcall (see section alien-funcall (page 122).) In all other contexts, functions are represented by function pointer types: (* (function ...)).

system-area-pointer [*Alien type*]

A pointer which is represented in Lisp as a system-area-pointer object (see section 6.5, page 102.)

8.2.4 The C-Call Package

The c-call package exports these type-equivalents to the C type of the same name: char, short, int, long, unsigned-char, unsigned-short, unsigned-int, unsigned-long, float, double. c-call also exports these types:

void [*Alien type*]

This type is used in function types to declare that no useful value is returned. Evaluation of an alien-funcall form will return zero values.

c-string [*Alien type*]

This type is similar to (* char), but is interpreted as a null-terminated string, and is automatically converted into a Lisp string when accessed. If the pointer is C NULL (or 0), then accessing gives Lisp nil.

Assigning a Lisp string to a c-string structure field or variable stores the contents of the string to the memory already pointed to by that variable. When an Alien of type (* char) is assigned to a c-string, then the c-string pointer is assigned to. This allows c-string pointers to be initialized. For example:

```
(def-alien-type nil (struct foo (str c-string)))

(defun make-foo (str) (let ((my-foo (make-alien (struct foo))))
(setf (slot my-foo 'str) (make-alien char (length str))) (setf (slot
my-foo 'str) str) my-foo))
```

Storing Lisp nil writes C NULL to the c-string pointer.

8.3 Alien Operations

This section describes the basic operations on Alien values.

8.3.1 Alien Access Operations

alien:deref *pointer-or-array* &rest*indices* [*Function*]

This function returns the value pointed to by an Alien pointer or the value of an Alien array element. If a pointer, an optional single index can be specified to give the equivalent of C pointer arithmetic; this index is scaled by the size of the type pointed to. If an array, the number of indices must be the same as the number of dimensions in the array type. deref can be set with setf to assign a new value.

alien:slot *struct-or-union slot-name* [*Function*]

This function extracts the value of slot *slot-name* from the an Alien struct or union. If *struct-or-union* is a pointer to a structure or union, then it is automatically dereferenced. This can be set with setf to assign a new value. Note that *slot-name* is evaluated, and need not be a compile-time constant (but only constant slot accesses are efficiently compiled.)

8.3.2 Alien Coercion Operations

alien:addr *alien-expr* [*Macro*]

This macro returns a pointer to the location specified by *alien-expr*, which must be either an Alien variable, a use of deref, a use of slot, or a use of extern-alien (page 120).

alien:cast *alien new-type* [*Macro*]

This macro converts *alien* to a new Alien with the specified *new-type*. Both types must be an Alien pointer, array or function type. Note that the result is not eq to the argument, but does refer to the same data bits.

alien:sap-alien *sap type* [*Macro*]

alien:alien-sap *alien-value* [*Function*]

sap-alien converts *sap* (a system area pointer see section 6.5, page 102) to an Alien value with the specified *type*. *type* is not evaluated.

alien-sap returns the SAP which points to *alien-value*'s data.

The *type* to sap-alien and the type of the *alien-value* to alien-sap must some Alien pointer, array or record type.

8.3.3 Alien Dynamic Allocation

Dynamic Aliens are allocated using the malloc library, so foreign code can call free on the result of make-alien, and Lisp code can call free-alien on objects allocated by foreign code.

alien:make-alien *type* {*size*} [*Macro*]

This macro returns a dynamically allocated Alien of the specified *type* (which is not evaluated.) The allocated memory is not initialized, and may contain arbitrary junk. If supplied, *size* is an expression to evaluate to compute the size of the allocated object. There are two major cases:

- When *type* is an array type, an array of that type is allocated and a *pointer* to it is returned. Note that you must use deref to change the result to an array before you can use deref to read or write elements:

  ```
  (defvar *foo* (make-alien (array char 10)))

  (type-of *foo*) ⇒ (alien (* (array (signed 8) 10)))

  (setf (deref (deref foo) 0) 10) ⇒ 10
  ```

  If supplied, *size* is used as the first dimension for the array.

- When *type* is any other type, then then an object for that type is allocated, and a *pointer* to it is returned. So (make-alien int) returns a (* int). If *size* is specified, then a block of that many objects is allocated, with the result pointing to the first one.

alien:free-alien *alien* [*Function*]

This function frees the storage for *alien* (which must have been allocated with make-alien or malloc.)

See also with-alien (page 119), which stack-allocates Aliens.

8.4 Alien Variables

Both local (stack allocated) and external (C global) Alien variables are supported.

8.4.1 Local Alien Variables

alien:with-alien {*(name type* {*initial-value*} *)*}* {*form*}* [*Macro*]

This macro establishes local alien variables with the specified Alien types and names for dynamic extent of the body. The variable *names* are established as symbol-macros; the bindings have lexical scope, and may be assigned with setq or setf. This form is analogous to defining a local variable in C: additional storage is allocated, and the initial value is copied.

with-alien also establishes a new scope for named structures and unions. Any *type* specified for a variable may contain name structure or union types with the slots specified. Within the lexical scope of the binding specifiers and body, a locally defined structure type *foo* can be referenced by its name using:

```
(struct foo)
```

8.4.2 External Alien Variables

External Alien names are strings, and Lisp names are symbols. When an external Alien is represented using a Lisp variable, there must be a way to convert from one name syntax into the other. The macros extern-alien, def-alien-variable and def-alien-routine (page 123) use this conversion heuristic:

- Alien names are converted to Lisp names by uppercasing and replacing underscores with hyphens.
- Conversely, Lisp names are converted to Alien names by lowercasing and replacing hyphens with underscores.
- Both the Lisp symbol and Alien string names may be separately specified by using a list of the form:

  (*alien-string lisp-symbol*)

alien:def-alien-variable *name type* [*Macro*]

This macro defines *name* as an external Alien variable of the specified Alien *type*. *name* and *type* are not evaluated. The Lisp name of the variable (see above) becomes a global Alien variable in the Lisp namespace. Global Alien variables are effectively "global symbol macros"; a reference to the variable fetches the contents of the external variable. Similarly, setting the variable stores new contents—the new contents must be of the declared *type*.

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-alien-variable "errno" int)

;; Now it is possible to get the value of the C variable errno simply by
;; referencing that Lisp variable:
;;
(print errno)
```

alien:extern-alien *name type* [*Macro*]

This macro returns an Alien with the specified *type* which points to an externally defined value. *name* is not evaluated, and may be specified either as a string or a symbol. *type* is an unevaluated Alien type specifier.

8.5 Alien Data Structure Example

Now that we have Alien types, operations and variables, we can manipulate foreign data structures. This C declaration can be translated into the following Alien type:

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

 ≡

(def-alien-type nil
  (struct foo
    (a int)
    (b (array (* (struct foo)) 100))))
```

With this definition, the following C expression can be translated in this way:

```
struct foo f;
f.b[7].a

 ≡

(with-alien ((f (struct foo)))
  (slot (deref (slot f 'b) 7) 'a)
  ;;
  ;; Do something with f...
  )
```

Or consider this example of an external C variable and some accesses:

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

extern struct c_struct *my_struct;

my_struct->x++;
my_struct->a = 5;
my_struct = my_struct->n;
```

which can be made be manipulated in Lisp like this:

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

(def-alien-variable "my_struct" (* c-struct))

(incf (slot my-struct 'x))
(setf (slot my-struct 'a) 5)
(setq my-struct (slot my-struct 'n))
```

8.6 Loading Unix Object Files

Foreign object files are loaded into the running Lisp process by load-foreign. First, it runs the linker on the files and libraries, creating an absolute Unix object file. This object file is then loaded into into the currently running Lisp. The external symbols defining routines and variables are made available for future external references (e.g. by extern-alien.) load-foreign must be run before any of the defined symbols are referenced.

Note that if a Lisp core image is saved (using save-lisp (page 18)), all loaded foreign code is lost when the image is restarted.

alien:load-foreign *files* &key :libraries :base-file :env [*Function*]

files is a simple-string or list of simple-strings specifying the names of the object files. *libraries* is 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). *base-file* is the file to use for the initial symbol table information. The default is the Lisp start up code: 'path: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.

8.7 Alien Function Calls

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.

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.

8.7.1 The alien-funcall Primitive

alien:alien-funcall *alien-function* &rest *arguments* [*Function*]

This function is the foreign function call primitive: *alien-function* is called with the supplied *arguments* and its value is returned. The *alien-function* is an arbitrary run-time expression; to call a constant function, use extern-alien (page 120) or def-alien-routine.

The type of *alien-function* must be (alien (function ...)) or (alien (* (function ...))), See section 8.2.3, page 117. The function type is used to determine how to call the function (as though it was declared with a prototype.) The type need not be known at compile time, but only known-type calls are efficiently compiled. Limitations:

- Structure type return values are not implemented.
- Passing of structures by value is not implemented.

Here is an example which allocates a (struct foo), calls a foreign function to initialize it, then returns a Lisp vector of all the (* (struct foo)) objects filled in by the foreign call:

```
;; Allocate a foo on the stack.
(with-alien ((f (struct foo)))
  ;;
  ;; Call some C function to fill in foo fields.
  (alien-funcall (extern-alien "mangle_foo" (function void (* foo)))
                 (addr f))
  ;;
  ;; Find how many foos to use by getting the A field.
  (let* ((num (slot f 'a))
         (result (make-array num)))
    ;;
    ;; Get a pointer to the array so that we don't have to keep extracting it:
    (with-alien ((a (* (array (* (struct foo)) 100)) (addr (slot f 'b))))
      ;;
      ;; Loop over the first N elements and stash them in the result vector.
      (dotimes (i num)
        (setf (svref result i) (deref (deref a) i)))
      result)))
```

8.7.2 The def-alien-routine Macro

alien:def-alien-routine *name result-type* {(*aname atype* {*style*})}* [*Macro*]

This macro is a convenience for automatically generating Lisp interfaces to simple foreign functions. The primary feature is the parameter style specification, which translates the C pass-by-reference idiom into additional return values.

name is usually a string external symbol, but may also be a symbol Lisp name or a list of the foreign name and the Lisp name. If only one name is specified, the other is automatically derived, (see section 8.4.2, page 120.)

result-type is the Alien type of the return value. Each remaining subform specifies an argument to the foreign function. *aname* is the symbol name of the argument to the constructed function (for documentation) and *atype* is the Alien type of corresponding foreign argument. The semantics of the actual call are the same as for alien-funcall (page 122). *style* should be one of the following:

:in
: specifies that the argument is passed by value. This is the default. :in arguments have no corresponding return value from the Lisp function.

:out
: specifies a pass-by-reference output value. The type of the argument must be a pointer to a fixed sized object (such as an integer or pointer). :out and :in-out cannot be used with pointers to arrays, records or functions. An object of the correct size is allocated, and its address is passed to the foreign function. When the function returns, the contents of this location are returned as one of the values of the Lisp function.

:copy
: is similar to :in, but the argument is copied to a pre-allocated object and a pointer to this object is passed to the foreign routine.

:in-out
: is 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 foreign routine. On return, the contents of this location is returned as an additional value.

Any efficiency-critical foreign interface function should be inline expanded by preceding def-alien-routine with:

```
(declaim (inline lisp-name))
```

In addition to avoiding the Lisp call overhead, this allows pointers, word-integers and floats to be passed using non-descriptor representations, avoiding consing (see section 5.11.2, page 85.)

8.7.3 def-alien-routine Example

Consider the C function cfoo with the following calling convention:

```
/* a for update
 * i out
 */
void cfoo (char *str, char *a, int *i);
```

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

```
(def-alien-routine "cfoo" void
  (str c-string)
  (a char :in-out)
  (i int :out))
```

The Lisp function cfoo will have two arguments (*str* and *a*) and two return values (*a* and *i*).

8.7.4 Calling Lisp from C

Calling Lisp functions from C is sometimes possible, but is rather hackish. See funcall0 ... funcall3 in the 'lisp/arch.h'. The arguments must be valid CMUCL object descriptors (e.g. fixnums must be left-shifted by 2.) See 'compiler/generic/objdef.lisp' or the derived file 'lisp/internals.h' for details of the object representation. 'lisp/internals.h' is mechanically generated, and is not part of the source distribution. It is distributed in the 'docs/' directory of the binary distribution.

Note that the garbage collector moves objects, and won't be able to fix up any references in C variables, so either turn GC off or don't keep Lisp pointers in C data unless they are to statically allocated objects. You can use purify (page 18) to place live data structures in static space so that they won't move during GC.

8.7.5 Accessing Lisp Arrays

Due to the way CMUCL manages memory, the amount of memory that can be dynamically allocated by malloc or make-alien (page 119) is limited[1].

To overcome this limitation, it is possible to access the content of Lisp arrays which are limited only by the amount of physical memory and swap space available. However, this technique is only useful if the foreign function takes pointers to memory instead of allocating memory for itself. In latter case, you will have to modify the foreign functions.

This technique takes advantage of the fact that CMUCL has specialized array types (see section 5.11.8, page 89) that match a typical C array. For example, a (simple-array double-float (100)) is stored in memory in essentially the same way as the C array double x[100] would be. The following function allows us to get the physical address of such a Lisp array:

```
(defun array-data-address (array)
  "Return the physical address of where the actual data of an array is
stored.

ARRAY must be a specialized array type in CMUCL.  This means ARRAY
must be an array of one of the following types:

                  double-float
                  single-float
                  (unsigned-byte 32)
                  (unsigned-byte 16)
                  (unsigned-byte  8)
                  (signed-byte 32)
                  (signed-byte 16)
                  (signed-byte  8)
"
  (declare (type (or #+signed-array (array (signed-byte 8))
                     #+signed-array (array (signed-byte 16))
                     #+signed-array (array (signed-byte 32))
                     (array (unsigned-byte 8))
                     (array (unsigned-byte 16))
                     (array (unsigned-byte 32))
                     (array single-float)
                     (array double-float))
                 array)
           (optimize (speed 3) (safety 0))
           (ext:optimize-interface (safety 3)))
  ;; with-array-data will get us to the actual data.  However, because
  ;; the array could have been displaced, we need to know where the
  ;; data starts.
```

[1] CMUCL mmaps a large piece of memory for its own use and this memory is typically about 8 MB above the start of the C heap. Thus, only about 8 MB of memory can be dynamically allocated.

```
(lisp::with-array-data ((data array)
                        (start)
                        (end))
  (declare (ignore end))
  ;; DATA is a specialized simple-array.  Memory is laid out like this:
  ;;
  ;;   byte offset    Value
  ;;        0         type code (should be 70 for double-float vector)
  ;;        4         4 * number of elements in vector
  ;;        8         1st element of vector
  ;;      ...         ...
  ;;
  (let ((addr (+ 8 (logandc1 7 (kernel:get-lisp-obj-address data))))
        (type-size (let ((type (array-element-type data)))
                     (cond ((or (equal type '(signed-byte 8))
                                (equal type '(unsigned-byte 8)))
                            1)
                           ((or (equal type '(signed-byte 16))
                                (equal type '(unsigned-byte 16)))
                            2)
                           ((or (equal type '(signed-byte 32))
                                (equal type '(unsigned-byte 32)))
                            4)
                           ((equal type 'single-float)
                            4)
                           ((equal type 'double-float)
                            8)
                           (t
                            (error "Unknown specialized array element type"))))))
    (declare (type (unsigned-byte 32) addr)
             (optimize (speed 3) (safety 0) (ext:inhibit-warnings 3)))
    (system:int-sap (the (unsigned-byte 32)
                      (+ addr (* type-size start)))))))
```

Assume we have the C function below that we wish to use:

```
double dotprod(double* x, double* y, int n)
{
  int k;
  double sum = 0;

  for (k = 0; k < n; ++k) {
    sum += x[k] * y[k];
  }
}
```

The following example generates two large arrays in Lisp, and calls the C function to do the desired computation. This would not have been possible using malloc or make-alien since we need about 16 MB of memory to hold the two arrays.

```
(def-alien-routine "dotprod" double
  (x (* double-float) :in)
  (y (* double-float) :in)
  (n int :in))

(let ((x (make-array 1000000 :element-type 'double-float))
      (y (make-array 1000000 :element-type 'double-float)))
```

```
;; Initialize X and Y somehow
(let ((x-addr (system:int-sap (array-data-address x)))
      (y-addr (system:int-sap (array-data-address y))))
  (dotprod x-addr y-addr 1000000)))
```

In this example, it may be useful to wrap the inner let expression in an unwind-protect that first turns off garbage collection and then turns garbage collection on afterwards. This will prevent garbage collection from moving x and y after we have obtained the (now erroneous) addresses but before the call to dotprod is made.

8.8 Step-by-Step Alien 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 'test.c':

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

struct c_struct *c_function (i, s, r, a)
    int i;
    char *s;
    struct c_struct *r;
    int a[10];
{
  int j;
  struct c_struct *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 c_struct *) malloc (sizeof(struct c_struct));
  r2->x = i + 5;
  r2->s = "A C string";
  return(r2);
};
```

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

```
;;; -*- Package: test-c-call -*-
(in-package "TEST-C-CALL")
(use-package "ALIEN")
(use-package "C-CALL")

;;; Define the record c-struct in Lisp.
(def-alien-type nil
    (struct c-struct
            (x int)
            (s c-string)))

;;; Define the Lisp function interface to the C routine.  It returns a
;;; pointer to a record of type c-struct.  It accepts four parameters:
```

```
;;; i, an int; s, a pointer to a string; r, a pointer to a c-struct
;;; record; and a, a pointer to the array of 10 ints.
;;;
;;; The INLINE declaration eliminates some efficiency notes about heap
;;; allocation of Alien values.
(declaim (inline c-function))
(def-alien-routine c-function
    (* (struct c-struct))
  (i int)
  (s c-string)
  (r (* (struct c-struct)))
  (a (array int 10)))

;;; A function which sets up the parameters to the C function and
;;; actually calls it.
(defun call-cfun ()
  (with-alien ((ar (array int 10))
               (c-struct (struct c-struct)))
    (dotimes (i 10)                    ; Fill array.
      (setf (deref ar i) i))
    (setf (slot c-struct 'x) 20)
    (setf (slot c-struct 's) "A Lisp String")

    (with-alien ((res (* (struct c-struct))
                      (c-function 5 "Another Lisp String" (addr c-struct) ar)))
      (format t "Returned from C function.~%")
      (multiple-value-prog1
          (values (slot res 'x)
                  (slot res 's))
        ;;
        ;; Deallocate result  after we are done using it.
        (free-alien res)))))
```

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

```
cc -c test.c
```

In order to enable incremental loading with some linkers, you may need to say:

```
cc -G 0 -c test.c
```

Once the C code has been compiled, you can start up Lisp and load it in:

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

;;; Compile the Lisp file.  This step can be done separately.  You don't have
;;; to recompile every time.
* (compile-file "test.lisp")

;;; Load the foreign object file to define the necessary symbols.  This must
;;; be done before loading any code that refers to these symbols.  next block
;;; of comments are actually the output of LOAD-FOREIGN.  Different linkers
;;; will give different warnings, but some warning about redefining the code
;;; size is typical.
* (load-foreign "test.o")
```

```
;;; Running library:load-foreign.csh...
;;; Loading object file...
;;; Parsing symbol table...
Warning:  "_gp" moved from #x00C082C0 to #x00C08460.
Warning:  "end" moved from #x00C00340 to #x00C004E0.

;;; o.k. now load the compiled Lisp object file.
* (load "test")

;;; Now we can 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. Under X, the output goes to the window in which Lisp was started; on a terminal, the output will overwrite the Hemlock screen image; in a Hemlock slave, standard output is '/dev/null' by default, so any output is discarded.

Chapter 9

Interprocess Communication under LISP

by William Lott and Bill Chiles

CMUCL offers a facility for interprocess communication (IPC) on top of using Unix system calls and the complications of that level of IPC. There is a simple remote-procedure-call (RPC) package build on top of TCP/IP sockets.

9.1 The REMOTE Package

The remote package provides simple RPC facility including interfaces for creating servers, connecting to already existing servers, and calling functions in other Lisp processes. The routines for establishing a connection between two processes, create-request-server and connect-to-remote-server, return *wire* structures. A wire maintains the current state of a connection, and all the RPC forms require a wire to indicate where to send requests.

9.1.1 Connecting Servers and Clients

Before a client can connect to a server, it must know the network address on which the server accepts connections. Network addresses consist of a host address or name, and a port number. Host addresses are either a string of the form VANCOUVER.SLISP.CS.CMU.EDU or a 32 bit unsigned integer. Port numbers are 16 bit unsigned integers. Note: *port* in this context has nothing to do with Mach ports and message passing.

When a process wants to receive connection requests (that is, become a server), it first picks an integer to use as the port. Only one server (Lisp or otherwise) can use a given port number on a given machine at any particular time. This can be an iterative process to find a free port: picking an integer and calling create-request-server. This function signals an error if the chosen port is unusable. You will probably want to write a loop using handler-case, catching conditions of type error, since this function does not signal more specific conditions.

wire:create-request-server *port* &optional *on-connect* [*Function*]

create-request-server sets up the current Lisp to accept connections on the given port. If port is unavailable for any reason, this signals an error. When a client connects to this port, the acceptance mechanism makes a wire structure and invokes the *on-connect* function. Invoking this function has a couple purposes, and *on-connect* may be nil in which case the system foregoes invoking any function at connect time.

The *on-connect* function is both a hook that allows you access to the wire created by the acceptance mechanism, and it confirms the connection. This function takes two arguments, the wire and the host address of the connecting process. See the section on host addresses below. When *on-connect* is nil, the request server allows all connections. When it is non-nil, the function returns two values, whether to accept the connection and a function the system should call when the connection terminates. Either value may be nil, but when the first value is nil, the acceptance mechanism destroys the wire.

create-request-server returns an object that destroy-request-server uses to terminate a connection.

wire:destroy-request-server *server* [*Function*] destroy-request-server takes the result of create-request-server and terminates that server. Any existing connections remain intact, but all additional connection attempts will fail.

wire:connect-to-remote-server *host port* &optional *on-death* [*Function*]

connect-to-remote-server attempts to connect to a remote server at the given *port* on *host* and returns a wire structure if it is successful. If *on-death* is non-nil, it is a function the system invokes when this connection terminates.

9.1.2 Remote Evaluations

After the server and client have connected, they each have a wire allowing function evaluation in the other process. This RPC mechanism has three flavors: for side-effect only, for a single value, and for multiple values.

Only a limited number of data types can be sent across wires as arguments for remote function calls and as return values: integers inclusively less than 32 bits in length, symbols, lists, and *remote-objects* (see section 9.1.3, page 131). The system sends symbols as two strings, the package name and the symbol name, and if the package doesn't exist remotely, the remote process signals an error. The system ignores other slots of symbols. Lists may be any tree of the above valid data types. To send other data types you must represent them in terms of these supported types. For example, you could use prin1-to-string locally, send the string, and use read-from-string remotely.

wire:remote *wire* {*call-specs*}* [*Macro*]

The remote macro arranges for the process at the other end of *wire* to invoke each of the functions in the *call-specs*. To make sure the system sends the remote evaluation requests over the wire, you must call wire-force-output.

Each of *call-specs* looks like a function call textually, but it has some odd constraints and semantics. The function position of the form must be the symbolic name of a function. remote evaluates each of the argument subforms for each of the *call-specs* locally in the current context, sending these values as the arguments for the functions.

Consider the following example:

```
(defun write-remote-string (str)
  (declare (simple-string str))
  (wire:remote wire
    (write-string str)))
```

The value of str in the local process is passed over the wire with a request to invoke write-string on the value. The system does not expect to remotely evaluate str for a value in the remote process.

wire:wire-force-output *wire* [*Function*]

wire-force-output flushes all internal buffers associated with *wire*, sending the remote requests. This is necessary after a call to remote.

wire:remote-value *wire call-spec* [*Macro*]

The remote-value macro is similar to the remote macro. remote-value only takes one *call-spec*, and it returns the value returned by the function call in the remote process. The value must be a valid type the system can send over a wire, and there is no need to call wire-force-output in conjunction with this interface.

If client unwinds past the call to remote-value, the server continues running, but the system ignores the value the server sends back.

If the server unwinds past the remotely requested call, instead of returning normally, remote-value returns two values, nil and t. Otherwise this returns the result of the remote evaluation and nil.

wire:remote-value-bind *wire* ({*variable*}*) remote-form {*local-forms*}* [*Macro*] remote-value-bind is similar to multiple-value-bind except the values bound come from *remote-form*'s evaluation in the remote process. The *local-forms* execute in an implicit progn.

If the client unwinds past the call to remote-value-bind, the server continues running, but the system ignores the values the server sends back.

If the server unwinds past the remotely requested call, instead of returning normally, the *local-forms* never execute, and remote-value-bind returns nil.

9.1.3 Remote Objects

The wire mechanism only directly supports a limited number of data types for transmission as arguments for remote function calls and as return values: integers inclusively less than 32 bits in length, symbols, lists. Sometimes it is useful to allow remote processes to refer to local data structures without allowing the remote process to operate on the data. We have *remote-objects* to support this without the need to represent the data structure in terms of the above data types, to send the representation to the remote process, to decode the representation, to later encode it again, and to send it back along the wire.

You can convert any Lisp object into a remote-object. When you send a remote-object along a wire, the system simply sends a unique token for it. In the remote process, the system looks up the token and returns a remote-object for the token. When the remote process needs to refer to the original Lisp object as an argument to a remote call back or as a return value, it uses the remote-object it has which the system converts to the unique token, sending that along the wire to the originating process. Upon receipt in the first process, the system converts the token back to the same (eq) remote-object.

wire:make-remote-object *object* [*Function*]

make-remote-object returns a remote-object that has *object* as its value. The remote-object can be passed across wires just like the directly supported wire data types.

wire:remote-object-p *object* [*Function*]

The function remote-object-p returns t if *object* is a remote object and nil otherwise.

wire:remote-object-local-p *remote* [*Function*]

The function remote-object-local-p returns t if *remote* refers to an object in the local process. This is can only occur if the local process created *remote* with make-remote-object.

wire:remote-object-eq *obj1 obj2* [*Function*]

The function remote-object-eq returns t if *obj1* and *obj2* refer to the same (eq) lisp object, regardless of which process created the remote-objects.

wire:remote-object-value *remote* [*Function*]

This function returns the original object used to create the given remote object. It is an error if some other process originally created the remote-object.

wire:forget-remote-translation *object* [*Function*]

This function removes the information and storage necessary to translate remote-objects back into *object*, so the next gc can reclaim the memory. You should use this when you no longer expect to receive references to *object*. If some remote process does send a reference to *object*, remote-object-value signals an error.

9.1.4 Host Addresses

The operating system maintains a database of all the valid host addresses. You can use this database to convert between host names and addresses and vice-versa.

ext:lookup-host-entry *host* [*Function*]

lookup-host-entry searches the database for the given *host* and returns a host-entry structure for it. If it fails to find *host* in the database, it returns nil. *Host* is either the address (as an integer) or the name (as a string) of the desired host.

ext:host-entry-name *host-entry* [*Function*]
ext:host-entry-aliases *host-entry* [*Function*]
ext:host-entry-addr-list *host-entry* [*Function*]
ext:host-entry-addr *host-entry* [*Function*]

host-entry-name, host-entry-aliases, and host-entry-addr-list each return the indicated slot from the host-entry structure. host-entry-addr returns the primary (first) address from the list returned by host-entry-addr-list.

9.2 The WIRE Package

The wire package provides for sending data along wires. The remote package sits on top of this package. All data sent with a given output routine must be read in the remote process with the complementary fetching routine. For example, if you send so a string with wire-output-string, the remote process must know to use wire-get-string. To avoid rigid data transfers and complicated code, the interface supports sending *tagged* data. With tagged data, the system sends a tag announcing the type of the next data, and the remote system takes care of fetching the appropriate type.

When using interfaces at the wire level instead of the RPC level, the remote process must read everything sent by these routines. If the remote process leaves any input on the wire, it will later mistake the data for an RPC request causing unknown lossage.

9.2.1 Untagged Data

When using these routines both ends of the wire know exactly what types are coming and going and in what order. This data is restricted to the following types:

- 8 bit unsigned bytes.
- 32 bit unsigned bytes.
- 32 bit integers.
- simple-strings less than 65535 in length.

wire:wire-output-byte *wire byte* [*Function*]
wire:wire-get-byte *wire* [*Function*]
wire:wire-output-number *wire number* [*Function*]
wire:wire-get-number *wire* &optional *signed* [*Function*]
wire:wire-output-string *wire string* [*Function*]
wire:wire-get-string *wire* [*Function*]

These functions either output or input an object of the specified data type. When you use any of these output routines to send data across the wire, you must use the corresponding input routine interpret the data.

9.2.2 Tagged Data

When using these routines, the system automatically transmits and interprets the tags for you, so both ends can figure out what kind of data transfers occur. Sending tagged data allows a greater variety of data types: integers inclusively less than 32 bits in length, symbols, lists, and *remote-objects* (see section 9.1.3, page 131). The system sends symbols as two strings, the package name and the symbol name, and if the package doesn't exist remotely, the remote process signals an error. The system ignores other slots of symbols. Lists may be any tree of the above valid data types. To send other data types you must represent them in terms of these supported types. For example, you could use prin1-to-string locally, send the string, and use read-from-string remotely.

wire:wire-output-object *wire object* &optional *cache-it* [*Function*]

wire:wire-get-object *wire* [*Function*]

The function wire-output-object sends *object* over *wire* preceded by a tag indicating its type.

If *cache-it* is non-nil, this function only sends *object* the first time it gets *object*. Each end of the wire associates a token with *object*, similar to remote-objects, allowing you to send the object more efficiently on successive transmissions. *cache-it* defaults to t for symbols and nil for other types. Since the RPC level requires function names, a high-level protocol based on a set of function calls saves time in sending the functions' names repeatedly.

The function wire-get-object reads the results of wire-output-object and returns that object.

9.2.3 Making Your Own Wires

You can create wires manually in addition to the remote package's interface creating them for you. To create a wire, you need a Unix *file descriptor*. If you are unfamiliar with Unix file descriptors, see section 2 of the Unix manual pages.

wire:make-wire *descriptor* [*Function*]

The function make-wire creates a new wire when supplied with the file descriptor to use for the underlying I/O operations.

wire:wire-p *object* [*Function*]

This function returns t if *object* is indeed a wire, nil otherwise.

wire:wire-fd *wire* [*Function*]

This function returns the file descriptor used by the *wire*.

9.3 Out-Of-Band Data

The TCP/IP protocol allows users to send data asynchronously, otherwise known as *out-of-band* data. When using this feature, the operating system interrupts the receiving process if this process has chosen to be notified about out-of-band data. The receiver can grab this input without affecting any information currently queued on the socket. Therefore, you can use this without interfering with any current activity due to other wire and remote interfaces.

Unfortunately, most implementations of TCP/IP are broken, so use of out-of-band data is limited for safety reasons. You can only reliably send one character at a time.

This routines in this section provide a mechanism for establishing handlers for out-of-band characters and for sending them out-of-band. These all take a Unix file descriptor instead of a wire, but you can fetch a wire's file descriptor with wire-fd.

wire:add-oob-handler *fd char handler* [*Function*]

The function add-oob-handler arranges for *handler* to be called whenever *char* shows up as out-of-band data on the file descriptor *fd*.

wire:remove-oob-handler *fd char* [*Function*]

This function removes the handler for the character *char* on the file descriptor *fd*.

wire:remove-all-oob-handlers *fd* [*Function*]

This function removes all handlers for the file descriptor *fd*.

wire:send-character-out-of-band *fd char* [*Function*]

This function Sends the character *char* down the file descriptor *fd* out-of-band.

Chapter 10

Debugger Programmer's Interface

The debugger programmers interface is exported from from the DEBUG-INTERNALS or DI package. This is a CMU extension that allows debugging tools to be written without detailed knowledge of the compiler or run-time system.

Some of the interface routines take a code-location as an argument. As described in the section on code-locations, some code-locations are unknown. When a function calls for a *basic-code-location*, it takes either type, but when it specifically names the argument *code-location*, the routine will signal an error if you give it an unknown code-location.

10.1 DI Exceptional Conditions

Some of these operations fail depending on the availability debugging information. In the most severe case, when someone saved a Lisp image stripping all debugging data structures, no operations are valid. In this case, even backtracing and finding frames is impossible. Some interfaces can simply return values indicating the lack of information, or their return values are naturally meaningful in light missing data. Other routines, as documented below, will signal serious-conditions when they discover awkward situations. This interface does not provide for programs to detect these situations other than by calling a routine that detects them and signals a condition. These are serious-conditions because the program using the interface must handle them before it can correctly continue execution. These debugging conditions are not errors since it is no fault of the programmers that the conditions occur.

10.1.1 Debug-conditions

The debug internals interface signals conditions when it can't adhere to its contract. These are serious-conditions because the program using the interface must handle them before it can correctly continue execution. These debugging conditions are not errors since it is no fault of the programmers that the conditions occur. The interface does not provide for programs to detect these situations other than calling a routine that detects them and signals a condition.

debug-condition [*Condition*]

This condition inherits from serious-condition, and all debug-conditions inherit from this. These must be handled, but they are not programmer errors.

no-debug-info [*Condition*]

This condition indicates there is absolutely no debugging information available.

no-debug-function-returns [*Condition*]

This condition indicates the system cannot return values from a frame since its debug-function lacks debug information details about returning values.

no-debug-blocks [*Condition*]

This condition indicates that a function was not compiled with debug-block information, but this information is necessary necessary for some requested operation.

no-debug-variables [*Condition*]

Similar to no-debug-blocks, except that variable information was requested.

lambda-list-unavailable [*Condition*]

Similar to no-debug-blocks, except that lambda list information was requested.

invalid-value [*Condition*]

This condition indicates a debug-variable has :invalid or :unknown value in a particular frame.

ambiguous-variable-name [*Condition*]

This condition indicates a user supplied debug-variable name identifies more than one valid variable in a particular frame.

10.1.2 Debug-errors

These are programmer errors resulting from misuse of the debugging tools' programmers' interface. You could have avoided an occurrence of one of these by using some routine to check the use of the routine generating the error.

debug-error [*Condition*]

This condition inherits from error, and all user programming errors inherit from this condition.

unhandled-condition [*Condition*]

This error results from a signalled debug-condition occurring without anyone handling it.

unknown-code-location [*Condition*]

This error indicates the invalid use of an unknown-code-location.

unknown-debug-variable [*Condition*]

This error indicates an attempt to use a debug-variable in conjunction with an inappropriate debug-function; for example, checking the variable's validity using a code-location in the wrong debug-function will signal this error.

frame-function-mismatch [*Condition*]

This error indicates you called a function returned by preprocess-for-eval on a frame other than the one for which the function had been prepared.

10.2 Debug-variables

Debug-variables represent the constant information about where the system stores argument and local variable values. The system uniquely identifies with an integer every instance of a variable with a particular name and package. To access a value, you must supply the frame along with the debug-variable since these are particular to a function, not every instance of a variable on the stack.

debug-variable-name *debug-variable* [*Function*]

This function returns the name of the *debug-variable*. The name is the name of the symbol used as an identifier when writing the code.

debug-variable-package *debug-variable* [*Function*]

This function returns the package name of the *debug-variable*. This is the package name of the symbol used as an identifier when writing the code.

debug-variable-symbol *debug-variable* [*Function*]

This function returns the symbol from interning debug-variable-name in the package named by debug-variable-package.

debug-variable-id *debug-variable* [*Function*]

This function returns the integer that makes *debug-variable*'s name and package name unique with respect to other *debug-variable*'s in the same function.

debug-variable-validity *debug-variable basic-code-location* [*Function*]

This function returns three values reflecting the validity of *debug-variable*'s value at *basic-code-location*:

:valid The value is known to be available.

:invalid The value is known to be unavailable.

:unknown The value's availability is unknown.

debug-variable-value *debug-variable frame* [*Function*]

This function returns the value stored for *debug-variable* in *frame*. The value may be invalid. This is SETF'able.

debug-variable-valid-value *debug-variable frame* [*Function*]

This function returns the value stored for *debug-variable* in *frame*. If the value is not :valid, then this signals an invalid-value error.

10.3 Frames

Frames describe a particular call on the stack for a particular thread. This is the environment for name resolution, getting arguments and locals, and returning values. The stack conceptually grows up, so the top of the stack is the most recently called function.

top-frame, frame-down, frame-up, and frame-debug-function can only fail when there is absolutely no debug information available. This can only happen when someone saved a Lisp image specifying that the system dump all debugging data.

top-frame [*Function*]

This function never returns the frame for itself, always the frame before calling top-frame.

frame-down *frame* [*Function*]

This returns the frame immediately below *frame* on the stack. When *frame* is the bottom of the stack, this returns nil.

frame-up *frame* [*Function*]

This returns the frame immediately above *frame* on the stack. When *frame* is the top of the stack, this returns nil.

frame-debug-function *frame* [*Function*]

This function returns the debug-function for the function whose call *frame* represents.

frame-code-location *frame* [*Function*]

This function returns the code-location where *frame*'s debug-function will continue running when program execution returns to *frame*. If someone interrupted this frame, the result could be an unknown code-location.

frame-catches *frame* [*Function*]

This function returns an a-list for all active catches in *frame* mapping catch tags to the code-locations at which the catch re-enters.

eval-in-frame *frame form* [*Function*]

This evaluates *form* in *frame*'s environment. This can signal several different debug-conditions since its success relies on a variety of inexact debug information: invalid-value, ambiguous-variable-name, frame-function-mismatch. See also preprocess-for-eval (page 139).

10.4 Debug-functions

Debug-functions represent the static information about a function determined at compile time—argument and variable storage, their lifetime information, etc. The debug-function also contains all the debug-blocks representing basic-blocks of code, and these contains information about specific code-locations in a debug-function.

do-debug-function-blocks (*block-var debug-function* {*result-form*}) {*form*}* [*Macro*]

This executes the forms in a context with *block-var* bound to each debug-block in *debug-function* successively. *Result-form* is an optional form to execute for a return value, and do-debug-function-blocks returns nilif there is no *result-form*. This signals a no-debug-blocks condition when the *debug-function* lacks debug-block information.

debug-function-lambda-list *debug-function* [*Function*]

This function returns a list representing the lambda-list for *debug-function*. The list has the following structure:

```
(required-var1 required-var2
...
(:optional var3 suppliedp-var4)
(:optional var5)
...
(:rest var6) (:rest var7)
...
(:keyword keyword-symbol var8 suppliedp-var9)
(:keyword keyword-symbol var10)
...
)
```

Each var*n* is a debug-variable; however, the symbol :deleted appears instead whenever the argument remains unreferenced throughout *debug-function*.

If there is no lambda-list information, this signals a lambda-list-unavailable condition.

do-debug-function-variables (*var debug-function* {*result*}) {*form*}* [*Macro*]

This macro executes each *form* in a context with *var* bound to each debug-variable in *debug-function*. This returns the value of executing *result* (defaults to nil). This may iterate over only some of *debug-function*'s variables or none depending on debug policy; for example, possibly the compilation only preserved argument information.

debug-variable-info-available *debug-function* [*Function*]

This function returns whether there is any variable information for *debug-function*. This is useful for distinguishing whether there were no locals in a function or whether there was no variable information. For example, if do-debug-function-variables executes its forms zero times, then you can use this function to determine the reason.

debug-function-symbol-variables *debug-function symbol* [*Function*]

This function returns a list of debug-variables in *debug-function* having the same name and package as *symbol*. If *symbol* is uninterned, then this returns a list of debug-variables without package names and with the same name as *symbol*. The result of this function is limited to the availability of variable information in *debug-function*; for example, possibly *debug-function* only knows about its arguments.

ambiguous-debug-variables *debug-function name-prefix-string* [*Function*]

This function returns a list of debug-variables in *debug-function* whose names contain *name-prefix-string* as an initial substring. The result of this function is limited to the availability of variable information in *debug-function*; for example, possibly *debug-function* only knows about its arguments.

preprocess-for-eval *form basic-code-location* [*Function*]

This function returns a function of one argument that evaluates *form* in the lexical context of *basic-code-location*. This allows efficient repeated evaluation of *form* at a certain place in a function which could be useful for conditional breaking. This signals a no-debug-variables condition when the code-location's debug-function has no debug-variable information available. The returned function takes a frame as an argument. See also eval-in-frame (page 138).

function-debug-function *function* [*Function*]

This function returns a debug-function that represents debug information for *function*.

debug-function-kind *debug-function* [*Function*]

This function returns the kind of function *debug-function* represents. The value is one of the following:

:optional — This kind of function is an entry point to an ordinary function. It handles optional defaulting, parsing keywords, etc.

:external — This kind of function is an entry point to an ordinary function. It checks argument values and count and calls the defined function.

:top-level — This kind of function executes one or more random top-level forms from a file.

:cleanup — This kind of function represents the cleanup forms in an unwind-protect.

nil — This kind of function is not one of the above; that is, it is not specially marked in any way.

debug-function-function *debug-function* [*Function*]

This function returns the Common Lisp function associated with the *debug-function*. This returns nil if the function is unavailable or is non-existent as a user callable function object.

debug-function-name *debug-function* [*Function*]

This function returns the name of the function represented by *debug-function*. This may be a string or a cons; do not assume it is a symbol.

10.5 Debug-blocks

Debug-blocks contain information pertinent to a specific range of code in a debug-function.

do-debug-block-locations (*code-var debug-block* {*result*}) {*form*}* [*Macro*]

This macro executes each *form* in a context with *code-var* bound to each code-location in *debug-block*. This returns the value of executing *result* (defaults to nil).

debug-block-successors *debug-block* [*Function*]

This function returns the list of possible code-locations where execution may continue when the basic-block represented by *debug-block* completes its execution.

debug-block-elsewhere-p *debug-block* [*Function*]

This function returns whether *debug-block* represents elsewhere code. This is code the compiler has moved out of a function's code sequence for optimization reasons. Code-locations in these blocks are unsuitable for stepping tools, and the first code-location has nothing to do with a normal starting location for the block.

10.6 Breakpoints

A breakpoint represents a function the system calls with the current frame when execution passes a certain code-location. A break point is active or inactive independent of its existence. They also have an extra slot for users to tag the breakpoint with information.

make-breakpoint *hook-function what* &key :kind :info :function-end-cookie [*Function*]

This function creates and returns a breakpoint. When program execution encounters the breakpoint, the system calls *hook-function*. *hook-function* takes the current frame for the function in which the program is running and the breakpoint object.

what and *kind* determine where in a function the system invokes *hook-function*. *what* is either a code-location or a debug-function. *kind* is one of :code-location, :function-start, or :function-end. Since the starts and ends of functions may not have code-locations representing them, designate these places by supplying *what* as a debug-function and *kind* indicating the :function-start or :function-end. When *what* is a debug-function and *kind* is :function-end, then hook-function must take two additional arguments, a list of values returned by the function and a function-end-cookie.

info is information supplied by and used by the user.

function-end-cookie is a function. To implement function-end breakpoints, the system uses starter breakpoints to establish the function-end breakpoint for each invocation of the function. Upon each entry, the system creates a unique cookie to identify the invocation, and when the user supplies a function for this argument, the system invokes it on the cookie. The system later invokes the function-end breakpoint hook on the same cookie. The user may save the cookie when passed to the function-end-cookie function for later comparison in the hook function.

This signals an error if *what* is an unknown code-location.

Note: Breakpoints in interpreted code or byte-compiled code are not implemented. Function-end breakpoints are not implemented for compiled functions that use the known local return convention (e.g. for block-compiled or self-recursive functions.)

activate-breakpoint *breakpoint* [*Function*]

This function causes the system to invoke the *breakpoint*'s hook-function until the next call to deactivate-breakpoint or delete-breakpoint. The system invokes breakpoint hook functions in the opposite order that you activate them.

deactivate-breakpoint *breakpoint* [*Function*]

This function stops the system from invoking the *breakpoint*'s hook-function.

breakpoint-active-p *breakpoint* [*Function*]

This returns whether *breakpoint* is currently active.

breakpoint-hook-function *breakpoint* [*Function*]

This function returns the *breakpoint*'s function the system calls when execution encounters *breakpoint*, and it is active. This is SETF'able.

breakpoint-info *breakpoint* [*Function*]

This function returns *breakpoint*'s information supplied by the user. This is SETF'able.

breakpoint-kind *breakpoint* [*Function*]

This function returns the *breakpoint*'s kind specification.

breakpoint-what *breakpoint* [*Function*]

This function returns the *breakpoint*'s what specification.

delete-breakpoint *breakpoint* [*Function*]

This function frees system storage and removes computational overhead associated with *breakpoint*. After calling this, *breakpoint* is useless and can never become active again.

10.7 Code-locations

Code-locations represent places in functions where the system has correct information about the function's environment and where interesting operations can occur—asking for a local variable's value, setting breakpoints, evaluating forms within the function's environment, etc.

Sometimes the interface returns unknown code-locations. These represent places in functions, but there is no debug information associated with them. Some operations accept these since they may succeed even with missing debug data. These operations' argument is named *basic-code-location* indicating they take known and unknown code-locations. If an operation names its argument *code-location*, and you supply an unknown one, it will signal an error. For example, frame-code-location may return an unknown code-location if someone interrupted Lisp in the given frame. The system knows where execution will continue, but this place in the code may not be a place for which the compiler dumped debug information.

code-location-debug-function *basic-code-location* [*Function*]

This function returns the debug-function representing information about the function corresponding to the code-location.

code-location-debug-block *basic-code-location* [*Function*]

This function returns the debug-block containing code-location if it is available. Some debug policies inhibit debug-block information, and if none is available, then this signals a no-debug-blocks condition.

code-location-top-level-form-offset *code-location* [*Function*]

This function returns the number of top-level forms before the one containing *code-location* as seen by the compiler in some compilation unit. A compilation unit is not necessarily a single file, see the section on debug-sources.

code-location-form-number *code-location* [*Function*]

This function returns the number of the form corresponding to *code-location*. The form number is derived by walking the subforms of a top-level form in depth-first order. While walking the top-level form, count one in depth-first order for each subform that is a cons. See form-number-translations (page 143).

code-location-debug-source *code-location* [*Function*]
This function returns *code-location*'s debug-source.

code-location-unknown-p *basic-code-location* [*Function*]
This function returns whether *basic-code-location* is unknown. It returns nilwhen the code-location is known.

code-location= *code-location1 code-location2* [*Function*]
This function returns whether the two code-locations are the same.

10.8 Debug-sources

Debug-sources represent how to get back the source for some code. The source is either a file (compile-file or load), a lambda-expression (compile, defun, defmacro), or a stream (something particular to CMUCL, compile-from-stream).

When compiling a source, the compiler counts each top-level form it processes, but when the compiler handles multiple files as one block compilation, the top-level form count continues past file boundaries. Therefore code-location-top-level-form-offset returns an offset that does not always start at zero for the code-location's debug-source. The offset into a particular source is code-location-top-level-form-offset minus debug-source-root-number.

Inside a top-level form, a code-location's form number indicates the subform corresponding to the code-location.

debug-source-from *debug-source* [*Function*]
This function returns an indication of the type of source. The following are the possible values:

:file from a file (obtained by compile-file if compiled).

:lisp from Lisp (obtained by compile if compiled).

:stream from a non-file stream (CMUCL supports compile-from-stream).

debug-source-name *debug-source* [*Function*]
This function returns the actual source in some sense represented by debug-source, which is related to debug-source-from:

:file the pathname of the file.

:lisp a lambda-expression.

:stream some descriptive string that's otherwise useless.

debug-source-created *debug-source* [*Function*]
This function returns the universal time someone created the source. This may be nil if it is unavailable.

debug-source-compiled *debug-source* [*Function*]
This function returns the time someone compiled the source. This is nilif the source is uncompiled.

debug-source-root-number *debug-source* [*Function*]
This returns the number of top-level forms processed by the compiler before compiling this source. If this source is uncompiled, this is zero. This may be zero even if the source is compiled since the first form in the first file compiled in one compilation, for example, must have a root number of zero—the compiler saw no other top-level forms before it.

10.9 Source Translation Utilities

These two functions provide a mechanism for converting the rather obscure (but highly compact) representation of source locations into an actual source form:

debug-source-start-positions *debug-source* [*Function*]

This function returns the file position of each top-level form a vector if *debug-source* is from a :file. If debug-source-from is :lisp or :stream, or the file is byte-compiled, then the result is nil.

form-number-translations *form tlf-number* [*Function*]

This function returns a table mapping form numbers (see code-location-form-number) to source-paths. A source-path indicates a descent into the top-level-form *form*, going directly to the subform corresponding to a form number. *tlf-number* is the top-level-form number of *form*.

source-path-context *form path context* [*Function*]

This function returns the subform of *form* indicated by the source-path. *Form* is a top-level form, and *path* is a source-path into it. *Context* is the number of enclosing forms to return instead of directly returning the source-path form. When *context* is non-zero, the form returned contains a marker, #:****HERE****, immediately before the form indicated by *path*.