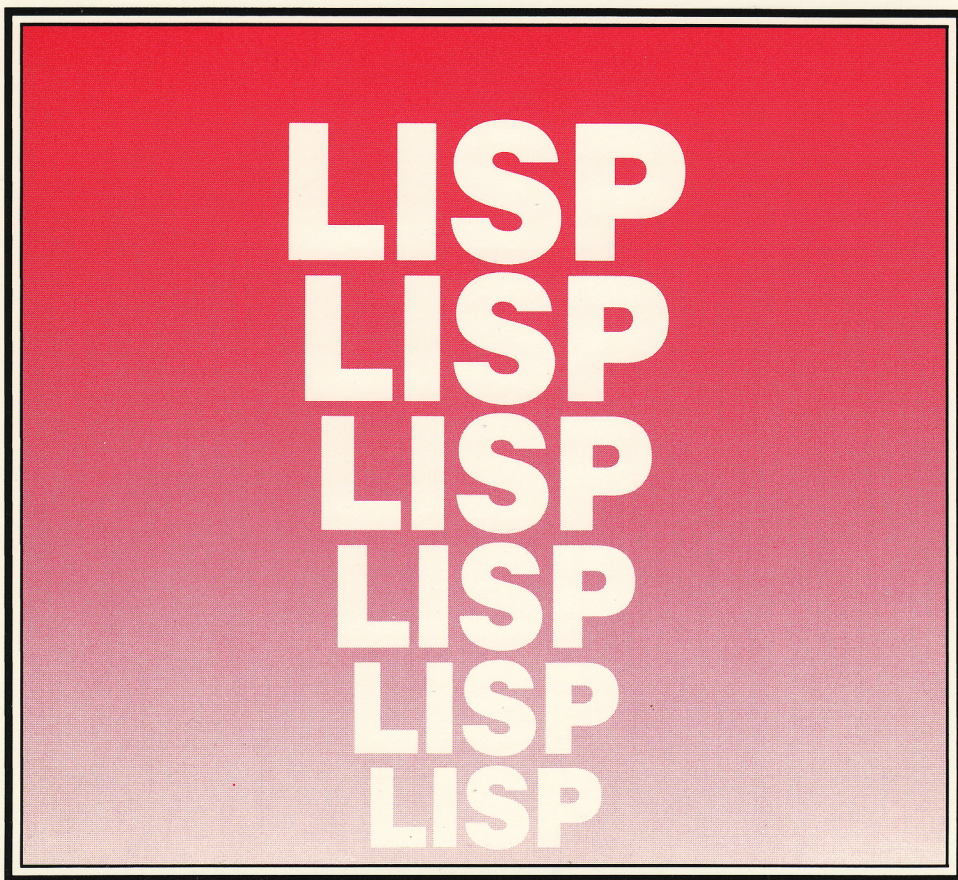


HP 9000 Series 300 Computers



## LISP Programmer's Guide



# **LISP Programmer's Guide**

## **for HP 9000 Series 300 Computers**

HP Part Number 98678-90040

© Copyright 1986 Hewlett-Packard Company

This document contains proprietary information which is protected by copyright. All rights are reserved. No part of this document may be photocopied, reproduced or translated to another language without the prior written consent of Hewlett-Packard Company. The information contained in this document is subject to change without notice.

#### Restricted Rights Legend

Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the Rights in Technical Data and Software clause in DAR 7-104.9(a).

Use of this manual and flexible disc(s) or tape cartridge(s) supplied for this pack is restricted to this product only. Additional copies of the programs can be made for security and back-up purposes only. Resale of the programs in their present form or with alterations, is expressly prohibited.

**Hewlett-Packard Company**

3404 East Harmony Road, Fort Collins, Colorado 80525

# Printing History

---

New editions of this manual will incorporate all material updated since the previous edition. Update packages may be issued between editions and contain replacement and additional pages to be merged into the manual by the user. Each updated page will be indicated by a revision date at the bottom of the page. A vertical bar in the margin indicates the changes on each page. Note that pages which are rearranged due to changes on a previous page are not considered revised.

The manual printing date and part number indicate its current edition. The printing date changes when a new edition is printed. (Minor corrections and updates which are incorporated at reprint do not cause the date to change.) The manual part number changes when extensive technical changes are incorporated.

March 1986...Edition 1

## NOTICE

The information contained in this document is subject to change without notice.

HEWLETT-PACKARD MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MANUAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE. Hewlett-Packard shall not be liable for errors contained herein or direct, indirect, special, incidental or consequential damages in connection with the furnishing, performance, or use of this material.

## WARRANTY

A copy of the specific warranty terms applicable to your Hewlett-Packard product and replacement parts can be obtained from your local Sales and Service Office.

# Table of Contents

---

## Chapter 1: Introduction

Purpose . . . . .	1
Audience . . . . .	1
Topics . . . . .	2
Conventions . . . . .	3
Fonts . . . . .	3
Function Descriptions . . . . .	3
Other Documentation . . . . .	4
NMODE Related Documents . . . . .	4
Lisp Documents . . . . .	4

## Chapter 2: Concepts

Introduction . . . . .	7
Scope and Extent . . . . .	8
Background . . . . .	8
Some Simple Scope Examples . . . . .	10
Shadowing . . . . .	11
Free Variables . . . . .	13
Scope and Extent of Symbols . . . . .	13
Special Bindings . . . . .	14
Closures . . . . .	18
Inside a Closure . . . . .	18
Example . . . . .	19
Listeners . . . . .	20
Listener Shorthands (Read Macros) . . . . .	21
Invoking a New Listener . . . . .	23
Listener Variables . . . . .	24
Listener Functions . . . . .	24
Garbage Collection . . . . .	25
The Heap . . . . .	25
When Garbage is Collected . . . . .	25
What the Garbage Collector Does . . . . .	26
Frequency and Duration . . . . .	27



Preprocessing . . . . .	29
So What? . . . . .	29
Example . . . . .	30
Compiling . . . . .	31
Macros . . . . .	31
Eval-When . . . . .	31

**Chapter 3: Programming Tips**

Introduction . . . . .	33
Speed Optimizations . . . . .	34
Constant Folding . . . . .	35
Safe Functional Transformations . . . . .	36
Unsafe Functional Transformations . . . . .	37
Inline Coding . . . . .	37
Safety . . . . .	37
Extensions . . . . .	38
Optimizing Wisely . . . . .	39
Using Heap Wisely . . . . .	41
Destructive Functions . . . . .	41
Shared List Structure . . . . .	42
Creating Public Functions . . . . .	44

**Chapter 4: Types and Declarations**

Introduction . . . . .	47
Available Types . . . . .	48
Type Specifiers . . . . .	49
Subset Type Specifiers . . . . .	50
Predicate Type Lists . . . . .	52
Defining Type Symbols . . . . .	53
Declarations . . . . .	56
Symbols for Declaring . . . . .	57
Symbols for Specifying Declarations . . . . .	60
Other Uses of Types . . . . .	65
Specifying the Type of a Form . . . . .	65
Checking Types . . . . .	66
Program Control With Types . . . . .	69
Type Coercion . . . . .	70
Examples of Declarations . . . . .	71
Global Declaration Examples . . . . .	71
Local Declaration Examples . . . . .	71

<b>Chapter 5: Macros</b>	
Introduction . . . . .	73
Background . . . . .	74
Examples . . . . .	78
Multiple-Value-Setf . . . . .	78
Substr . . . . .	79
<b>Chapter 6: Object-Oriented Programming</b>	
Introduction . . . . .	81
What's an Object? . . . . .	82
Messages . . . . .	83
Terms . . . . .	84
Getting Started . . . . .	85
Defining an Instance Type . . . . .	85
Defining a Method . . . . .	89
Creating an Instance . . . . .	90
Sending Messages . . . . .	90
A Brief Tutorial . . . . .	91
Another Short Example . . . . .	95
Inheritance . . . . .	97
Defining a Type that Inherits . . . . .	97
Inheriting Methods . . . . .	98
Inheriting Instance Variables . . . . .	103
Caveats . . . . .	107
Initialization . . . . .	108
Custom Initializations . . . . .	110
Universal Methods . . . . .	112
Equality Methods . . . . .	113
Checking the Type of Instances . . . . .	114
Copying Instances . . . . .	115
Redefining Instance Types . . . . .	116
Undefining Instance Types . . . . .	117
<b>Chapter 7: Calling Non-Lisp Routines</b>	
Introduction . . . . .	119
Background Information . . . . .	120
Object File Format . . . . .	120
Entry Points . . . . .	120
Loading Foreign Functions . . . . .	121
Load-related Variables and Functions . . . . .	123
Creating an Access Routine . . . . .	124
Parameter Specifiers . . . . .	125

Result Specifiers .....	130
Restrictions .....	130
Examples .....	130
Accessing Non-Lisp Variables .....	131
Example .....	132
Complete Examples .....	133
C .....	133
Pascal .....	137
Fortran .....	141
Assembly Language .....	143

## Chapter 8: Debugging Tools

Introduction .....	145
Concepts .....	146
The Execution Stack .....	146
Alternate Listener Modes .....	146
Compiled vs. Interpreted .....	147
Optimizations .....	147
The Execution Monitor .....	148
Execution Monitor Items .....	148
Breakpoints .....	149
Commands .....	149
Program Errors .....	152
Options .....	153
The Execution Stack Browser .....	154
Commands .....	154
The Inspector .....	156
Commands .....	157
Inspecting Instance Types .....	158
Example .....	158
Debug Listener Mode .....	160
Debug Listener Commands .....	160
Tracing a Function .....	164
Nested Trace Specifiers .....	166
Changing Options .....	166
Tracing Order .....	167
Examples .....	168
Miscellany .....	168
The Break Loop .....	169
Break Loop Related Variables .....	169

## **Chapter 9: File System Dependencies**

Introduction . . . . .	173
Pathnames . . . . .	174
Examples . . . . .	175
Resolving Filenames . . . . .	175
Loading Modules . . . . .	176
Multiple File Modules . . . . .	177

## **Chapter 10: Extensions**

Introduction . . . . .	177
System Functions . . . . .	178
Operating System Access Functions . . . . .	180
Error Signalling and Handling . . . . .	182
Defining Error Symbols . . . . .	182
Error Handling . . . . .	183
Error Handling Example . . . . .	188

## **Index**





# Introduction

---




## Purpose

Welcome to *The Lisp Programmer's Guide* for Hewlett-Packard's Lisp workstation. As its name suggests, this book provides the information necessary to write Common Lisp programs on this system. This book is about HP's implementation of the Common Lisp language and related utilities; you will not learn how to use the workstation by reading this book.


This book is organized so coverage of each topic is independent. For the most part, you do not have to read any prior chapters to understand what is presented in a particular chapter. However, in order to present useful comprehensive examples, there are minor degrees of interdependence between some of the examples.

---

## Audience






You do not have to be an advanced Lisp programmer to use this manual. You should have a firm grasp of Lisp fundamentals, and be familiar with general programming concepts. Most of the topics in this book are specific to Hewlett-Packard's Lisp; some of the material presented here is also in Steele's *Common Lisp*, but in a less accessible form. Some details given in Steele have been omitted from this book to make the presentation less cluttered.



---

## Topics

The following chapters comprise this manual:

<b>Chapter 1 Introduction</b>	The very same introduction you are reading at this very moment!	
<b>Chapter 2 Concepts</b>	Explains some important Lisp programming concepts such as scope and extent, garbage collection, and listeners.	
<b>Chapter 3 Programming Tips</b>	Contains advice on how to write efficient Lisp code.	
<b>Chapter 4 Types and Declarations</b>	Discusses the data types available in Common Lisp, how to define your own types, and use of declarations to enhance efficiency.	
<b>Chapter 5 Macros</b>	Explains and demonstrates the Common Lisp facility for defining macros.	
<b>Chapter 6 Object-Oriented Programming</b>	Describes object-oriented programming and the Lisp constructs available to facilitate it. Several short examples are presented.	
<b>Chapter 7 Calling Non-Lisp Routines</b>	Discusses how to call functions written in another language from Lisp.	
<b>Chapter 8 Debugging Tools</b>	Explains the various tools available to help you debug your Lisp programs.	
<b>Chapter 9 File System Dependencies</b>	Describes how Hewlett-Packard has implemented Common Lisp pathnames to fit HP-UX, as well as how HP has defined several system-dependent functions.	
<b>Chapter 10 Extensions</b>	Describes the functions that Hewlett-Packard has added to our implementation of Common Lisp. This chapter does not cover extensions that are described elsewhere (such as extensions for debugging and object-oriented programming).	

---

# Conventions



## Fonts


Various typographical conventions are used throughout this manual.

- File names, and Lisp symbols and values are printed in a typewriter font (e.g., `/users/gurus/ritchie`, `car`, `56`). All Lisp code appears in this font also.
- Bold font is used to highlight new terms when they are defined, to stress important sections, and to represent keystroke commands (e.g., **Don't touch that dial!**, **C-X C-F.**)
- The names of HP-UX commands, the names of manuals, and parameters in the description of functions, macros, and methods, appear in italics (e.g., *vi*, the *NMODE Users Guide*, *assoc-list*).

## Function Descriptions


Here is an example of how this manual describes the syntax of calls to functions, macros, and special-forms.

`(deftype type-name lambda-list {declaration |doc-string}* {form}*)` *Macro*



The basic form is ( followed by the name of the function, followed by a description of the form of the arguments, followed by ) and an indication of whether it is a variable, function, macro, or special form. Descriptions of arguments use the following notation:

- Names in italics (e.g. *declaration*) identify where actual arguments should appear. What those arguments should be is usually described in the following paragraph.
- { and } are used as syntactic parentheses to indicate grouping of arguments.
- Zero or more repetitions of the preceding argument or group of arguments are indicated by \*.
- One or more repetitions of the preceding argument or group of arguments are indicated by +.
- Two arguments or groups of arguments separated by a | indicate a choice of either of the two arguments.
- Arguments surrounded by brackets [ ] are optional.
- In descriptions of functions, *&optional*, *&key*, and *&rest* indicate that the parameters following them fall into that particular class. For instance, in `(foo x &key :radix)`, `foo` has one required parameter `x`, and will accept a `:radix` keyword parameter.



Descriptions of methods for object-oriented programming are similar, except that they are not bracketed by parentheses.



---

## Other Documentation

When using this manual, you may occasionally be pointed to other documentation that was shipped with your Lisp system. For reference, here is a list of the other documentation and what it contains.

### NMODE Related Documents

#### Reference Manual

- *NMODE Command Reference*

There are several appendices which provide reference information on the commands that are available in the different parts of the system. These include: Quick Reference, Complete Command Reference, Browser menu commands and softkey commands.

#### Technique Manual

- *NMODE User's Guide*

This manual explains the NMODE user environment. The chapters give general information about command syntaxes, displayed information and general rules of operation for the browsers and text buffers. Simple customizations and extensions to the system are also explained.

### Lisp Documents

#### Reference Manuals

- *Common Lisp* by Guy Steele

This is a purchased manual provided with the documentation package. It describes the Common Lisp standard. The “Bible” of Common Lisp.

- *Lisp Quick Reference*

This is a “mini-reference” for Common Lisp. The forms, functions, macros, and others are arranged alphabetically, and cross-referenced by functionality. HP extensions are included.

- *Lisp Language Reference*

This is an alphabetical reference of Common Lisp and HP extensions.

## Lisp Techniques

- *Lisp* by Patrick H. Winston

This is a purchased manual provided with the documentation package. It is an introduction to programming in Common Lisp.

- *Lisp: A Gentle Introduction to Symbolic Computing* by David Touretzky

This is a purchased manual provided with the documentation package. It provides a very clear explanation of the fundamental concepts of Lisp. Note that its examples are not written in Common Lisp.

The *Gentle Introduction* is provided for non-programmers, perhaps your manager, who want to know “what’s this all about”.

# Notes





## Introduction

Lisp is substantially different from mainstream programming languages like Pascal and C. Even if you are a veteran programmer in another language, there are probably some concepts that will be new to you. Since some of these concepts are subtle, understanding them is key to your success as a Lisp programmer. This chapter covers a number of important things about Lisp. It discusses

- Common Lisp's scoping rules
- Closures (function objects)
- Listeners (read-eval-print loops)
- Garbage collection
- Preprocessing
- A few important things about compiling Lisp code.



---

# Scope and Extent

## Background

Before introducing scope and extent, it is necessary to define a few terms.

- A **name** is a means of identifying something. It is often important to distinguish between the name of something and the thing itself. In Common Lisp, names are strings of characters, such as `x` or `dog`. Names can identify symbols and variables. When a name, such as `dog`, appears in this book, it refers to the thing named `dog`. The words “the name `dog`” would be used to refer to the name itself.
- **Symbols** are Common Lisp data objects. They have several components, including a print name, a home package, a global value, a global function definition, and a property list.
- A **variable** is a slot for storing a value (which can usually be any Common Lisp data object).
- The association between a name and a variable, or a name and a symbol is a **binding**. A name is said to be **bound** to a variable if the name is currently associated with that variable.
- A **closure** is the combination of a function and an environment. The environment portion of a closure maintains the bindings that are in effect for that closure. Closures allow bindings to persist even when the function that established them is no longer executing. See the “Closure” section later in this section for more information.

All programming languages must have rules that resolve exactly what a name is bound to. These rules are necessary because not every entity has a unique name. You must be able to use the same name in different contexts to refer to different entities. On the other hand, you may sometimes want to have a particular name always refer to the same entity. To fully understand a programming language, you must understand the rules that determine what entity a name identifies.

Two concepts central to these rules are **scope** and **extent**. The scope of an object is the textual area of a program in which references to that object may occur. If someone asked you what the scope of a particular variable is, you could answer with a list of the line numbers where that variable can be referenced.

Extent is more elusive. The extent of an object is the period of time in which an object can be referred to. Here time is usually described in terms like “during the execution of this function.”

**Example:** Consider the function,

```
(defun kung (x)
  (cond ((atom x) "It's an atom")
        (t "No such luck")))
```

The scope of `x` is the function `kung`. The scope is the textual area in which we can use the name `x` to refer to whatever object was the first argument to `kung`.

In this example, the extent of `x` is the duration of the execution of `kung`. Unfortunately, this is not the general case. To simplify an explanation of scope and extent in Common Lisp, we need to define a few more terms that describe some possible scopes and extents.

- Entities with **lexical scope** can be referenced by name only within the textual area of the construct that established that entity. For instance, the function `kung` establishes a binding between the name `x` and its first parameter, and that binding may only be referred to within the body of `kung`.
- An entity that has **indefinite scope** can be referred to anywhere. Common Lisp symbols have indefinite scope.
- Things that have **indefinite extent** exist as long as it is possible to reference them. The binding of the name `x` in `kung` has indefinite extent. In this case, the extent of the binding of `x` is only the duration of the execution of `kung`. As soon as `kung` finishes executing, there is no way of accessing the binding since it has lexical scope and the name `x` does not identify the parameter outside of `kung`. However, if a function creates a closure, then the bindings of its parameters may still be accessible after the function has finished executing.
- An entity with **dynamic extent** can be referenced any time between when it is established and when it is disestablished. Entities are established by the execution of some language construct. When that construct terminates in any way, the entities it established are disestablished.
- The term **dynamic scope** is sometimes used when discussing things that have indefinite scope and dynamic extent.

## Some Simple Scope Examples

Now that we have a framework for talking about scope and extent, we can look at some more examples of how they work in Common Lisp.

- Bindings between variables and names have lexical scope and indefinite extent.

```
(let ((x 1) y)
  (setq y 100)
  (if (evenp y) (setq x (+ x 1))))
```

In this example, the `let` form binds the names `x` and `y` to their respective variables which can be referred to by these names anywhere in the body of the `let`. `x` and `y` have lexical scope and indefinite extent, but in this case, there is no way to refer to the variables after the `let` has completed.

Here is an example where the extent of the bindings of lexical variables is longer than the the duration of the form that established them. It creates a closure that allows the bindings to persist.

```
(setq funny-adder (let ((x 6))
  #'(lambda (z)
    (if (zerop z) (setq x 0)
        (+ x z))))
)

(funcall funny-adder 7) ⇒ 13
(funcall funny-adder 0) ⇒ 0
(funcall funny-adder 7) ⇒ 7
```

The `let` returns a closure as the value of `funny-adder`. The closure's environment contains the binding of the variable `x`. This closure retains the ability to access and modify the variable `x`. After the value of `x` is modified by the second `funcall`, the value returned by `(funcall funny-adder 7)` is different.

- Symbols have indefinite scope and indefinite extent.

```
(let ((x 1))
  (setq y (+ x 6)))
```

In the `let` form above, the name `y` refers to the symbol `y`. The global value cell of the symbol `y` is set to `(+ x 6)`. In this case, `y` is called a **free variable**, because it is not bound within the scope of the `let`.

- Special variable bindings have indefinite scope and dynamic extent.

```
(defun sample ()
  (let ((x 45)) (declare (special x))
    (sample2)))

(defun sample2 ()
  (+ x 92))
```

The `let` in `sample` establishes a special binding of `x`. This binding can be referenced anywhere (like it is in `sample2`) until evaluation of the `let` completes. After that, the binding is disestablished and can no longer be referenced.

## Shadowing

### Lexical Shadowing

The rules we have discussed so far are slightly inadequate. What happens when, within the scope of a lexical variable, we create a new binding for the name of that variable?

```
(defun foo (x y)
  (print x)          ; This x is the parameter
  (let ((x (+ y 1))) ; y is the parameter y
    (print x)       ; This x is the let variable
    (print x)       ; This x is the parameter
```

In `foo` we have two variables named `x` — the first is one of the formal parameters of `foo`; the second is established by the `let` form. When the name `x` occurs, how do we know which of the two variables it identifies? The answer: It identifies the most recent binding that is still in effect.

When the first `(print x)` form occurs, `x` is still bound to the first parameter of `foo`. The second `(print x)` however, occurs inside the body of the `let`, which has established a new variable and bound it to the name `x`. The third `(print x)` occurs outside the body of the `let`, but still within the function `foo`, so it refers to the parameter `x`.

`(foo 2 3) ⇒ 2` and prints `2 4 2`

In situations such as this, we say that the inner binding of `x` **shadows** its outer binding.

Global symbols may be lexically shadowed. The following example demonstrates shadowing the global symbol `car`.

```
(defun stupid-tuna (alist)
  (flet ((car (1) (cdr 1)))
    (car alist)))

(stupid-tuna '(a b c)) ⇒ (B C)
```

The function `car` established by the `flet`, shadows the global function definition of the symbol `car`.

The above examples demonstrate lexical shadowing. It is also possible to have dynamic shadowing of things with indefinite scope and dynamic extent.

### Dynamic Shadowing

Catchers established by a `catch` special form have dynamic extent. It is possible for a function that establishes a catcher to call another function that establishes a catcher with the same name. When there is more than one active catcher with the same name, any `throw` to that name will go to the most recently established catcher.

```
(defun foo1 (x)
  (catch 'disc (* 2 (foo2 x))))

(defun foo2 (x)
  (catch 'disc (+ 1 (foo3 x))))

(defun foo3 (x)
  (throw 'disc x))
```

In this example, the catcher `disc` established in `foo1` is dynamically shadowed by the catcher established in `foo2`, so the `throw` from `foo3` will be caught by the `catch` in `foo2`.

```
(foo1 6) ⇒ 12
```

When this is evaluated, `foo2` returns the value it caught, `x`, and `foo1` returns this value times 2. Now if we changed `foo2` so that the catcher established in it had another name, say `football`, then the `throw` from `foo3` would be caught by the `catch` form in `foo1`.

```
(defun foo2 (x)
  (catch 'football (+ 1 (foo3 x))))
```

In this case,

```
(foo1 6) ⇒ 6
```

## Free Variables

A variable is **free** if it is not lexically bound. For example,

```
(let ((x 3) (y 9))
  (setq z (+ x y)))
```

In this form, `z` is a free variable. A free variable refers to the global value cell of the symbol with the same name. A bound variable refers to the global value cell of the symbol if the binding of that variable is declared to be **special**.

## Scope and Extent of Symbols

The bindings between names and symbols have indefinite scope and indefinite extent. After a symbol is established, it can be referenced anywhere its name is not shadowed.

```
(setq x '(a b c)) ; Establishes symbol x

(let (x)          ; Rebinds x for scope of let
  (setq x 13))   ; This is the local variable x

x ⇒ (a b c)

(let (y)
  (setq x 13))   ; The x referenced here is the symbol

x ⇒ 13
```

The first `let` above rebinds the name `x` to a local variable so the enclosed `setq` does not affect the symbol `x`. The second `let` however, does not rebind `x` and the `setq` changes the global value cell of the symbol.

## Special Bindings

In Common Lisp, you can declare variable bindings to be **special**. A special variable is bound to the global value cell of the symbol with the same name as the variable. Special variable bindings have indefinite scope and dynamic extent, which means they can be referenced anywhere they are not shadowed, from the time they are established until the establishing form terminates. After the establishing form terminates, **the global value cell of the affected symbol is restored to the value it had before the special binding was established**. This is what makes special variables “special”.

You can declare a variable to be special everywhere (with a `defvar`, or `proclaim`), or only in particular places. The following is an example of declaring a variable to be special in a particular place.

```
(setq x '(the original x))

(defun foo ()
  (let ((x 5)) (declare (special x))
    (foo2)
    x))

(defun foo2 ()      ; The free variable x refers to the
  (setq x (+ x 9))) ; special binding established in foo.

(foo) ⇒ 14
x ⇒ (THE ORIGINAL X)
```

The `setq` sets the global value of the symbol `x` to `(the original x)`. When `foo` is called, its `let` establishes a new binding for `x` that is declared to be special. The call to `foo2` occurs within the extent of the special binding of `x`, so any reference to `x` (as long as `x` is not lexically shadowed) is to the global value of the symbol `x` (which has been changed by the `let`). If the special declaration was omitted from `foo`, then an error would occur in `foo2` when it attempted to add 9 and the list `(the original x)`. When the `let` in `foo` terminates, the special binding is no longer in effect, and the previous value of the symbol `x` is restored.

Parameters to functions can be declared special.

```
(defun kung (x y) (declare (special x))
  (foo (+ y 7)))

(defun foo (z)
  (+ x y z))

(setq y 1000 x 5000)

(kung 5 6) ⇒ 1018
(foo 13) ⇒ 6013
```

This situation is analogous to the previous example. Since `kung` declares `x` to be special, when it calls `foo`, the `x` in `foo` refers to the new value of the symbol `x` (which is the value of the first argument to `kung`). The global value of `y` on the other hand, is not changed since the binding of the variable `y` is not declared special. Note that calling `foo` outside of `kung` with the same argument (13) gives different results because the value of `x` has been restored to 5000.

### Variables that are Always Special

When a variable is special everywhere, it is as if every new binding of the name of the variable is declared special.

There are two basic ways of making a variable be special everywhere. The macros `defvar` and `defparameter` are the preferred way, but proclaiming a variable to be special achieves the same effect.

```
(defvar *username* "Joe Schmo")
```

is equivalent to

```
(progn (proclaim '(special *username*))
  (set '*username* "Joe Schmo")
  '*username*)
```

Both establish a symbol named `*username*` whose bindings are always special. In system code, the names of global special variables that are exported usually begin and end with a `*` to avoid name conflicts.



Let's look at a few examples of globally special variables:

```
(defvar *indent-level* 0) ; Set up the special variable
;;;
;;; Print-list prints a list indented to show the structure
;;; of its nesting.
;;;
(defun print-list (l)
  (format t "~v,OT~A~%" *indent-level* "(")
  (dolist (element l)
    (typecase element
      (atom (format t "~v,OT~A~%" *indent-level* element))
      (list (let ((*indent-level* (+ 3 *indent-level*))) ; indent-level
              (print-list element)) ; is rebound
            )
    )
  )
  (format t "~v,OT~A~%" *indent-level* ")")
)
```

In this example, the special variable `*indent-level*` maintains the number of spaces to indent before printing. Before recursively calling `print-list`, `*indent-level*` is rebound to a higher number so that nested lists are indented more. When the `let` terminates after return from the recursive call, the global value of `*indent-level*` is reset to whatever it was before the `let`.

Here's another example:

```
(defvar z 1000)

(defun fun1 (z) ; A special binding of z
  (print (fun2))
  z)

(defun fun2 ()
  (let ((z (* z 2))) ; Another special binding of z
    z))

(fun1 12) ⇒ 12 after printing 24
z ⇒ 1000
```

`z` is declared to be a global special variable with initial value 1000 by the `defvar` form. `fun1` rebinds the value of `z` to the value of its parameter. This binding can be referred to anywhere, but only while `fun1` is executing.

The `let` form in `fun2` rebinds `z` yet another time. Note that the free variable `z` in `(* z 2)` refers to the binding established by `fun1` and not the original global value of `z`. When the `let` in `foo2` finishes, it returns `24` and the binding of `z` it established is undone, restoring the binding established by `fun1`.

`fun1` returns `12` (the value of its parameter) and `z` is reset to its original global value (`1000`).

The bindings of special variables can be thought of as a stack (though this is not necessarily how they are implemented). Each time a new special binding of a name is established, a new value holder is added to the top of the stack. When that binding is disestablished, the holder is popped off and the previous binding becomes the current one.

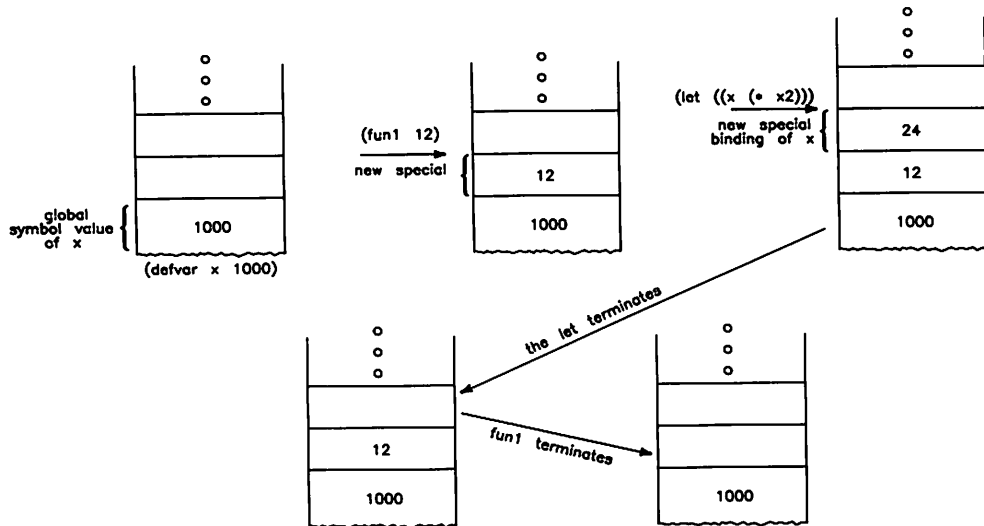


Figure 2-1. Conceptual Model of Special Variables

---

## Closures

Closures are similar to function objects in that they can be funcalled or applied just like functions. The difference is that a closure contains some knowledge of the lexical environment in which it was created.

A closure may be created by executing one of the following forms within a lexical environment, such as a `let` or a function:

```
#'(lambda ...  
#'x where x is defined by let or labels
```

A closure will be created if the `lambda`, or the definition of `x`, references any lexically visible entity (e.g. local variables, parameters, blocks, `tagbody` labels) other than special variables. If no such references exist, a normal function object is created.

### Inside a Closure

A closure is a compound object with two parts: a data part, and a function part. The function part is like a normal function object. The data part contains variables and control flow information. The variables are non-special variables which are referred to by the function part and were (are) locally bound within the scope in which the closure was created. The control flow information allows the closure to return from blocks or go to labels which were lexically visible where the closure was created.

### Upward and Downward

Closures are both **upwardly** and **downwardly** mobile. To be downwardly mobile, a closure needs to remain valid only as long as the scope in which it was created is still active. Many languages provide downwardly mobile closures; for example, HP Pascal procedure variables implement downwardly mobile closures. To be upwardly mobile, a closure needs to remain valid after the scope in which it was created has terminated.

In Common Lisp, closures are upwardly mobile with respect to lexical variables. Any local variables (including function parameters) which are referenced by closures continue to exist even after the scope in which they were bound has terminated.

Common Lisp closures are downwardly mobile with respect to `tagbody` labels and block names which were lexically visible where the closure was created. However, it is not possible to return from a block which has terminated or go to a label defined in a `tagbody` which has terminated.

## Compilation

The function part of a closure will be either interpreted or compiled, depending on whether the code that created it was interpreted or compiled (i.e. compiled code creates compiled closures). The function part of a closure is actually a part of the larger function in which it is nested.

## Example

One possible application of closures is to implement simple object-oriented programming capabilities. In this scheme, objects are closures, so the functions to create objects return closures. Here is a function to create a rectangle object. The variables that are being “wrapped up” in the closure are the length and width of the rectangle. The function part returns or modifies the values of these variables based on the keywords they are passed.

```
(defun make-rectangle (length width)
  ;; Return a closure
  #'(lambda (operation &rest args)
      ;; Any key in this CASE is the name of a method
      (case operation
        (:width width)
        (:length length)
        (:area (* width length))
        (:set-width (setf width (car args)))
        (:set-length (setf length (car args)))
        (:type 'rectangle)
        (:describe (format t
                           "A mini-object of type rectangle~%Width="A~%Length="A~%"
                           width
                           length))
      )
  )
)
```

To invoke operations on these “objects”, you funcall them with the name of the operation and any arguments required by those operations.

```
(defmacro my-send (object method-name &rest args)
  '(funcall ,object ,method-name ,@args)
)

(setf rect1 (make-rectangle 6 2))
(my-send rect1 :type) ⇒ RECTANGLE
(my-send rect1 :length) ⇒ 6
(my-send rect1 :set-length 10) ⇒ 10
(my-send rect1 :length) ⇒ 10
```

Other interesting applications of closures are discussed in *Structure and Interpretation of Computer Programs* by Harold Abelson and Gerald Jay Sussman with Julie Sussman.

---

## Listeners


A **listener** (also known as a **read-eval-print loop**) is a central part in your interaction with Lisp. It is the program that reads an input form, evaluates it, and prints the results. Usually, this default behavior is what you desire and there is no reason for you to be concerned about the listener. However, some situations benefit from specialized listener modes that provide additional features. For instance, debugging after an error occurs can be facilitated by a mode that simplifies access to information about the state of the Lisp system.

Additionally, a listener mode can provide shorthand notations for capabilities that are often used in that listener. A listener can be made to look to the user like a command interpreter specific to a particular situation, without foregoing the ability to do “normal” things like evaluate a Lisp form.

The five listener modes that are provided by HP’s Common Lisp system are:

- |         |  |
|---------|--|
| Lisp    | The default listener. Provides a few shorthands.   |
| Break   | Simple break loop. Entered when the <code>debug</code> module is not loaded, and an error is signalled or <code>break</code> is called. In this mode, the listener reads from and prints to the stream <code>*debug-io*</code> , even though the full debugger is not available.             |
| Debug   | Break loop when the <code>debug</code> module is loaded. Invoked when an error is signalled or the function <code>break</code> is called. In this mode, the listener also reads from and prints to the stream <code>*debug-io*</code> . Provides shorthands for calling debugging functions. |
| Inspect | Invoked from the function <code>inspect</code> . Provides shorthands for calling functions that access and display Lisp data objects.  |
| Monitor | Invoked from the <code>step</code> macro. Behavior is the same as Lisp mode, but the prompt reminds you that the execution monitor is active.  |


## Listener Shorthands (Read Macros)



To save typing and improve code readability, the reader part of a listener can interpret particular series of characters in distinctive ways. These character combinations are called **read macros**. An example of a read macro defined in Common Lisp is the single quote character (`'`). This tells the listener to replace the quoted form with a call to the `quote` special form. For example, the normal Lisp reader translates `'foo` into `(quote foo)`.

The Lisp listener modes like `debug` and `inspect` provide a set of shorthands (macros) for common operations (see the “Debugging Tools” chapter for details on those operations). These macros do nothing more than expand into an already documented form. They are only a typing aid for the interactive user. They are not recommended for use in code files. To see the expansion of a listener macro, simply evaluate the macro preceded by a single quote in the appropriate listener mode. Note that these shorthands **must be enabled** before they can be used.

### Enabling a Listener Macro Character



To enable them, you must first decide what single character or pair of characters you wish to use to access the built in macros. It is important that you pick a character or characters that will not interfere with other aspects of the system. The single quote for instance would be a tragic choice. See the table of standard character syntax types in the “Input/Output” chapter of Steele. The characters in that table marked with asterisks are good potential characters for accessing the listener shorthands. HP has picked one of those characters (`!` : the exclamation point) as the default listener macro character.


If you have chosen to use a single character (let's say `?`), then evaluating a form like

```
(system:on '(listener-read-macro #\?))
```

will enable that character as the listener macro character, so any form immediately preceded by it will be treated specially by the `debug` and `inspect` listeners. If you want to use `!` as the macro character, you can evaluate

```
(system:on 'listener-read-macro)
```

to enable it.



You can also use a sequence of two characters to access the listener macros. These are known as dispatch macros. The first character is a signal to the reader that it should treat the next character differently from usual. Here we discuss only the case where the first character is `#`. The `#` character is used as a dispatch character throughout Common Lisp (to read in octal or hex numbers for instance). If you decide to use two characters to access the listener macros, you should consult the Standard `#` Macro Character Syntax table in the “Input/Output” chapter of Steele to find a pair of characters that is not used for some other purpose.

Once you have selected a character to use as the second character of the dispatch macro, you must evaluate a form to enable it. Assume that you wanted to use #z to access the listener macros. The following form would be evaluated to put it into effect.

```
(system:on '(listener-read-dispatch-macro #\z))
```

To enable the default sequence of #!, you would evaluate

```
(system:on 'listener-read-dispatch-macro)
```

Advanced users who have defined their own dispatch macro character can use it as the first character in a listener macro sequence by calling `system:on` with a list (`listener-read-dispatch-macro dispatch-character char`).

### **Disabling a Listener Macro Character**

To deactivate a single character you enabled as the listener macro character, evaluate the form

```
(system:off 'listener-read-macro)
```

There is no need to specify the character.

To deactivate a pair of characters that you enabled, evaluate

```
(system:off 'listener-read-dispatch-macro)
```

### **Accessing Status of Listener Macro Characters**

You can determine the current listener read macro character, and the current dispatching macro characters with the function `system:on-off?`.

The function call

```
(sys:on-off? 'listener-read-macro)
```

returns the current listener read macro character if one is in effect, and `nil` if there is none in effect.

The function call

```
(sys:on-off? 'listener-read-dispatch-macro)
```

returns a list whose first element is the dispatching macro character and whose second element is the syntaxing character, or `nil` if there is no dispatching listener macro in effect.

## Invoking a New Listener

Most of the time, the system takes care of invoking the proper listener. When an error occurs, you get the debug listener; when you're inspecting a data object, you get the inspect listener. Sometime, though, you may want to invoke a listener yourself. The function `system:listener` is defined for that purpose.

`(system:listener mode banner &optional` *Function*  
    `reader`  
    `evaluator`  
    `printer`  
    `name`  
    `top-level?`)

The two required parameters are *mode* (a keyword) and *banner* (a string). The mode symbol name appears in the listener prompt. It is `LISP` for the normal top-level listener, `BREAK` for the simple break loop, `DEBUG` for the debugging break loop, `INSPECT` for the inspect function, and `MONITOR` for the execution monitor. When the listener is invoked, the string *banner* is printed to `*standard-output*`.

The first three optional parameters are the reader, evaluator, and printer to be used by the listener. Each is expected to be an object that can be "funcalled", preferably a symbol naming the appropriate function. These are called in order on each pass through the listener loop. They default to `system:reader`, `eval`, and `system:printer` respectively. With these values, the invoked listener will operate like the normal Common Lisp listener. The reader function is called with no arguments, the evaluator is called with what the reader returns (a single value), and the printer is called with one argument (if `eval` returns multiple values, the printer is called once for each value). Note that no stream argument is passed. If no values are returned by `eval`, the printer will not be called. The reader and printer are expected to use the appropriate stream. For instance, the stream `*debug-io*` is used instead of `*standard-input*` and `*standard-output*` when the listener mode is `debug`.

The parameter *name* is a string that will appear in the prompt before the mode.

The last optional parameter, *top-level?* indicates whether this listener should trap listener aborts. The function `system:listener-abort` throws back to the nearest listener for which *top-level?* is true. In a normal system, this will be the top level invocation of `system:listener`.



## Listener Variables

There are some special variables that affect and reflect the current listener.

`system:*exit-listener-on-eof*`

*Variable*

This variable controls the behavior of a listener when it reads an end of file. If `system:*exit-listener-on-eof*` is true, then when a reader reads an end of file, it exits. This will essentially cause the Lisp process to terminate. For interactive use, you should set `system:*exit-listener-on-eof*` to nil.

`system:*listener-mode*`

*Variable*

`system:*listener-banner*`

*Variable*

`system:*listener-read*`

*Variable*

`system:*listener-eval*`

*Variable*

`system:*listener-print*`

*Variable*

`system:*listener-name*`

*Variable*

These variables provide information about the current listener. For instance, `system:*listener-read*` stores the name of the current listener's read function.

## Listener Functions

There are a few functions in the `system` package that affect the current listener.

`(system:listener-abort)`

*Function*

Throws to the closest nested listener which was invoked with *top-level?* true. Sets `*applyhook*` and `*evalhook*` to nil.

`(system:listener-quit)`

*Function*

Quits the current listener by first exiting its read-eval-print loop and then throwing to the nearest enclosing error handler (most often the nearest enclosing listener).


`(system:listener-continue)`

*Function*

Continues from the current listener by exiting its read-eval-print loop. The function `system:listener` returns nil in this case.

---

## Garbage Collection




A feature of Lisp that contributes greatly to a programmer's productivity is automatic storage allocation. To create a new data object, one needs only to call the appropriate function (e.g. `cons`, `list`, `make-array`) and the storage for that object is automatically allocated from a central pool. Among other things, this makes it easier to write programs that are not limited to a specific maximum problem size.

However, this capability does not come free. Since the amount of virtual memory available to Lisp is not unlimited, the system must occasionally pause to “recycle” memory that has been used and then discarded. This process is known as **garbage collection** or **reclaiming**. Understanding garbage collection can reduce frustration and make you a better Lisp programmer.

### The Heap

The Lisp **heap** is a general-purpose storage area. It is used primarily for storing data objects (lists, arrays, bignums, etc.). Interpreted code and uninterned symbols also reside in the heap.

### When Garbage is Collected



When Lisp is first started, all the available heap space is in one large contiguous area. Space is allocated sequentially from this area. When only 40 000 bytes of free heap remain, a garbage collect is automatically invoked. Note that there are unused objects scattered throughout the heap. (If there are not, a garbage collect will not do any good.)

#### Full Heap

The heap is considered full when either

1. An attempt is made to allocate an object and two garbage collects have been invoked and there is still insufficient space to allocate the object.
2. Three consecutive garbage collects have resulted in very little free heap. This indicates that almost all of the CPU time is being spent garbage collecting and very little processing is getting done.

When the heap becomes full, a continuable error is signalled, putting you into a break loop or debug listener. For the duration of the break loop, you have available 40 000 bytes of heap. This allows a **very limited** amount of intervention to recover from the problem. Within the break loop you have two choices for recovery:

1. Quit out of the break loop with `!a` (i.e., call `sys:listener-abort`) or `!q` (i.e., call `sys:listener-abort`). This will abort the function that was attempting to allocate more heap.
2. Release enough heap so that the interrupted function may proceed to allocate the required space. This could be done by setting a global variable that contains some large data structure to `nil`. You may then continue the interrupted function with `!c` (i.e., call `sys:listener-continue`).

If a reclaim occurs during the break loop, no messages to that effect will be printed. If the heap becomes full during the break loop (according to the above definition of full), the Lisp process will be terminated.

## What the Garbage Collector Does

HP's garbage collector is a compacting garbage collector. To make the space taken by unused objects suitable for reallocation, the garbage collector needs to make the unused space contiguous, otherwise the space will be fragmented in chunks whose sizes may or may not be useful for reallocation. This means that after reclaiming, all the useful data is in a contiguous section of the heap that starts at the lowest heap address. All of the newly freed space is above this contiguous section. The collector will usually have to move some data to achieve this.

Consider a model of the heap before and after reclaiming.

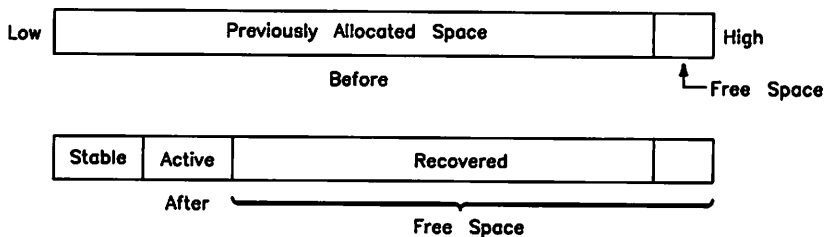


Figure 2-2. The LISP Heap

The three labeled areas are:

- stable        The number of useful data items that were not moved during reclaiming.
- active        The number of useful data items that were moved during reclaiming.
- recovered    The number of data items that were freed during reclaiming.

## Frequency and Duration

There are many factors which affect how often garbage collection occurs and how long it takes. The most important ones are

- Heap size
- Applications (such as the NMODE environment) being run
- System configuration (swap space, real memory, CPU, disk storage)

Let's look at each of these areas in more detail.

### Heap Size

This is an obvious factor in garbage collection. A larger heap will give you fewer reclaims since there is more space to be allocated. On the other hand, with a larger heap, reclaims will take longer since there is more heap for the garbage collector to deal with. Also, there will more likely be a lower percentage of the heap in real memory, so paging time will increase.

The "correct" heap size is a matter of personal preference, the application being run, and the HP-UX configuration. For any application, there is a some minimum size of heap that is required; beyond that there is some minimum sized heap that is required for reasonable response. Note that a large (bigger than six or seven megabytes) heap requires that your HP-UX swap space be configured to allow enough room for the larger Lisp process to run.

A big heap will make garbage collection longer, but less frequent. A smaller heap will have shorter, more frequent reclaims.

## Applications

The programs you run under Lisp affect how often and for how long garbage must be collected. If an application allocates heap often or in large amounts, the heap will be used up faster and thus reclaims will occur more frequently. If an application allocates a large amount of space that remains in use between reclaims, then garbage collection will take longer because more things must be moved.

The NMODE programming environment is an example of an application that is almost constantly allocating heap space. Every character you type into a buffer must have space allocated to it. If you keep more than a few buffers in memory, then the garbage collector will have to move a fair amount of data. That is why reclaims are more frequent and longer when NMODE is running than when you are just running a simple bare Common Lisp read-eval-print loop. This is not meant to discourage you from using NMODE or its facilities, but to give you an appreciation for why garbage needs to be collected more often when it is running. One minor thing that you can do in NMODE to free up space that would otherwise be wasted is to periodically clear out the OUTPUT buffer.


## System Configuration

The hardware that makes up your system affects the duration of reclaims. The factors that you should be aware of are

- |             |   |
|-------------|---|
| CPU         | A MC68020 processor has a faster clock speed, larger cache, and wider data paths than an MC68010. This means that a 320 system will collect garbage faster than a 310.  |
| Real Memory | The ratio of real (physical) memory to Lisp process size affects the length of garbage collects. The closer this ratio is to one, the less paging (a relatively slow procedure) will occur during reclaims.   |
| Swap Device | The speed of the disk used for paging affects reclaiming time. In general, a faster swap device leads to faster reclaims. For example, a typical Lisp system running on a configuration using a 7914 for swapping will reclaim slightly faster than a system that swaps on a 7945 or 7946. The speed improvement depends on how much swapping must be done. |

---


## Preprocessing



A key component of Hewlett-Packard's Common Lisp implementation is the preprocessor. Preprocessing is the first stage of translation for both interpreted and compiled code. In some sense, the preprocessor is itself a compiler, but it translates Lisp into an intermediate representation language instead of assembly or machine code. This intermediate language is then either executed interpretively or compiled.

### So What?

At first glance, the existence of the preprocessor may seem insignificant. However, it has at least two important ramifications:

- 
- The times at which certain things occur, such as the expansion of macros, affect the apparent semantics of your code. This is usually noticeable only if you nest top-level forms (see Steele, p. 66) within other constructs.
  - Since the preprocessor does much of the work that would normally be done by a compiler, interpreted and compiled code are more closely linked. This means that some idiosyncracies related to preprocess time are present in both interpreted and compiled code. With a more conventional Lisp implementation, these idiosyncracies would not show up until you compiled the code. One exception to this is `progn`, which behaves differently when compiled than when interpreted. This is to assure adherence to the Common Lisp standard which says "... if a `progn` form appears at top level, then all forms within that `progn` are considered by the compiler to be top-level forms."

A pathological example is necessary to characterize the kind of unexpected behavior that may occur if you nest forms that are usually used at top level, without considering exactly when certain things happen.

## Example

Consider the following code:

```
(defun weird ()
  (defmacro foo ()
    '(print "macro1"))
  (foo)
  (defun foo ()
    (print "function"))
  (foo)
  (defmacro foo ()
    '(print "macro2"))
  (foo)
  nil)

(weird)
(foo)
```

You might expect the call to `weird` to print

```
"macro1"
"function"
"macro2"
NIL
```


and the call to `foo` outside `weird` to print `"macro2"`. But they don't. What really happens is that `weird` prints

```
"macro1"
"macro1"
"macro2"
NIL
```

and the call to `foo` prints `"function"`. Why?

Macro definitions are established at preprocess time. Similarly, macro calls are expanded at preprocess time. The behavior of the first call to `foo` (within `weird`) makes sense: it calls the macro defined in the preceding form.

The behavior of the second call to `foo` does not make sense. You would expect it to call the `function` defined immediately before the call, but it doesn't. It doesn't because macro calls are expanded at preprocess time. When the preprocessor sees the second call to `foo`, it checks to see if there is a macro definition for it. There is, so it expands the call using the definition established by the first `defmacro` within `weird`.



The final call to `foo` within `weird` makes sense; it uses the macro definition established by the preceding `defmacro`. But why does the call to `foo` outside `weird` call the function instead of the macro we defined after the definition of the function? Because macro definitions are established at preprocess time while function definitions are established at eval time. So the ordering of the function and macro definitions for `foo` is insignificant: the `foo` macros get defined when `weird` is defined; the function `foo` gets defined when `weird` is called.


In general, you should not encounter many problems like this. But if you do have unexpected behavior in a program, examine your code for this kind of bug.

---

## Compiling

Once you've written a set of Lisp functions and macros that provide a particular functionality, you probably want to "package" that code into a loadable file. This is done with the `compile-file` function. In his chapter on packages (chapter 11), Steele describes the format of the source for a typical module. Beyond that, there are a few small things that are helpful to know when you are compiling your code.

### Macros



When you're compiling code that contains macro calls, the macros must be defined before they are referenced. This is because macro calls are expanded when code is preprocessed. If a macro is not defined before it is referenced, the call will be processed as if it was a function call. This is true even if the call is inside a `defun`.


For this reason, you should put macro definitions before function definitions in your source file.

### Eval-When

The `eval-when` special form controls when a series of forms gets evaluated. To understand why such a facility is needed, consider the action of the compiler when it reads the following top-level form from a source file.

```
(setq foo 12)
```

The difference between evaluating this form and compiling it is that the compiler does **not** set `foo` to 12; it generates code to set `foo` when the object file is loaded. What do you do if you want `foo` to be set in the compiler's environment, rather than at load time?



```
(eval-when (compile)
  (setq foo 12))
```

There are three situations that can be specified in an `eval-when`: `compile`, `load`, and `eval`.



## Example

Assume you have a program that has the following delay loop in it.

```
(dotimes (i *delay-amount*)
)
```

When you're running an interpreted version of the code, 500 is a good value for `*delay-amount*`, but when the code is compiled, you discover that the loop now runs much too fast. A solution is to set `*delay-amount*` inside an `eval-when`.

```
(eval-when (load)
  (defvar *delay-amount* 2000))
(eval-when (eval)
  (defvar *delay-amount* 500))
```

## Macros

If a source file defines some macros that are called **only** by functions defined in that same file, you can make your compiled file size smaller by having the macros defined only when the file is compiled or evaluated. This is done by putting the `defmacros` in the body of an `eval-when`.

```
;;; These macros are internal to this file
(eval-when (compile eval)

  ;; The foo macro
  (defmacro foo (x)
    :
  )
  :
)
```

## Implicit Eval-Whens

Calls to some Common Lisp functions, special forms, and macros are treated as if they are always within the body of an `eval-when` that specifies the `load`, `compile`, and `eval` situations. This behavior is consistent with the semantics of these forms. The affected forms are:

```
define-modify-macro
define-setf-method
defmacro
defsetf
defstruct
export
import
in-package
make-package
proclaim
rename-package
require
shadow
shadowing-import
unexport
unintern
unuse-package
use-package
```

# Programming Tips

---

## Introduction

For a given problem, there are many different programs that provide a solution to the problem. Obviously, some of these programs will be better than the others in terms of execution speed, and/or code size. How can you make sure that the Lisp programs you write perform well in these areas?

This chapter contains tips that will help you produce efficient Lisp code. The first part describes the kinds of optimizations that may be made and how to enable them. The second part discusses how to use these optimizations safely and effectively.

Before you can rush in and start writing these superfast Lisp programs, you need to know a little about the architecture of Hewlett-Packard's implementation of Common Lisp. The language system is made up of three basic components:

1. A preprocessor that converts Lisp code into an intermediate form. Most of the optimizations are done by the preprocessor. Use of the preprocessor also helps to insure semantic consistency between interpreted and compiled code.
2. An interpreter that interprets the output from the preprocessor.
3. A compiler that compiles preprocessor output into machine executable form.

Even though the preprocessor is used both when compiling and interpreting Lisp, its behavior is not exactly the same in both situations. When the preprocessor is called by the compiler, it bases its optimizations on the proclaimed or declared value of the `speed` quality. When called by the evaluator, the preprocessor bases its optimizations on the `extn:eval-speed` quality. Even if `extn:eval-speed` and `speed` have the same value, compiled code may be optimized more than interpreted code. For instance, the interpreter will check that the argument of a `the` actually is the specified type, while code emitted by the compiler will not.

As stated above, the preprocessor performs the transformations to optimize your Lisp code, so it is the component that we are most concerned with in this chapter. You can find out what modifications the preprocessor has made to your code with the function `extn:pp-expand`.

(`extn:pp-expand form &optional in-eval environ`)

Function

A call to `pp-expand` returns a form equivalent to the preprocessed version of *form*. If the optional argument *in-eval* is given and is non-nil, then the return value represents what the preprocessor would do to *form* when called from `eval` (i.e. the form is interpreted). Otherwise the value returned by `pp-expand` reflects the effects of the preprocessor when it is preprocessing forms for the compiler. Interpreted code may be less optimized. The optional argument *environ* is the environment you wish to have *form* expanded in (defaults to a null lexical environment). This is analogous to the optional environment argument to `macroexpand`.

---

## Speed Optimizations

Using the special form `declare` and the function `proclaim` will allow the preprocessor to optimize your code. (See the “Types” chapter for information on how to use `declare` and `proclaim`.) The preprocessor uses type information known at preprocess time as well as “general” declarations to optimize its output. Since they change what code is actually being executed, optimizations may affect error detection and handling. It may be more difficult to debug an optimized program for this reason.

Currently the preprocessor recognizes the `speed` and `safety` qualities in an `optimize` declaration or proclamation. The other qualities defined in Common Lisp (`space` and `compilation-speed`) are allowed but have no effect on the code that the preprocessor passes on to the interpreter or compiler.

HP has added a few other symbols for use in declarations and/or proclamations: `extn:system-lisp`, `extn:eval-speed`, and `extn:upward-closures`. The “Extensions” subsection later in this section describes exactly how to use these facilities.

The types of optimizations that the preprocessor performs are

- Constant folding: expressions whose values can be determined at preprocess time are replaced with that value if the functions in the expression have no side-effects.
- Conversion of `&keyword` and `&rest` parameters to positional parameters.
- Safe functional transformations: function calls are replaced with more efficient calls that always preserve the semantics of the program.

- Inline coding of function calls: the code that implements a function replaces the call, thus avoiding the overhead of a function call.
- Unsafe functional transforms: function calls are replaced with more efficient calls that may assume arguments of a particular type.

What each of these optimizations entails is described shortly. Which of them are performed depends on the values of `speed` or `extn:eval-speed`, and `extn:system-lisp`. The `speed` optimization level defaults to 1, `extn:eval-speed` to 0, and `extn:system-lisp` defaults to `nil`.

The following table summarizes the effects of various values of `speed` (for compiled code) and `extn:eval-speed` (for interpreted code) on preprocessor output:

- |   |  |
|---|--|
| 0 | No optimizations<br>No constant folding.<br>No functional transformations; all function calls are passed intact through the preprocessor.  |
| 1 | Some optimizations<br>No constant folding.<br>Safe functional transformations.<br>Conversion of <code>&amp;rest</code> and <code>&amp;keyword</code> parameters to positional parameters.<br>Open coding based on declarations.  |
| 2 | Level 1 optimizations.<br>Constant folding.  |
| 3 | Level 2 optimizations.<br>Optimization of structure operations: slot access functions and <code>self</code> methods are made inline with no type checking.<br>Additional functional transformations that either do less error checking than at speed 1, or significantly increase code size.<br>No checking for correct number of arguments in function calls. |

## Constant Folding

If a function has no side effects and all its arguments are preprocess-time constants, the result of the application of the function to its arguments will be inserted into the code instead of the function call.

## Safe Functional Transformations

Certain function calls may be transformed into different function calls or more efficient code sequences when doing so will not alter the semantics of Common Lisp. Transformations of this type fall into two broad categories:

1. Those which can be done unconditionally. Examples:

```
(typep x 'integer) ⇒ (integerp x) ⇒ inline tag check on x  
(+ x y z) ⇒ (lisp::binary-+ (lisp::binary-+ x y) z)  
(eq x y) ⇒ inline check for machine word equivalence
```

2. Those which can be done based on type declarations. Examples:

```
(car (the cons x)) ⇒ inline sequence without nil check  
(car (the list x)) ⇒ inline sequence with nil check
```

```
(let (a b)  
  (declare (fixnum a b))  
  (+ a b)) ⇒ fixnum + which assumes fixnum arguments, but  
             may return a bignum
```

```
(let (a b)  
  (declare (simple-string a b))  
  (when (string= a b) (print "ok"))) ⇒ Call to fast string= routine
```

```
(proclaim '(fixnum x))  
(+ x 3) ⇒ inline + which assumes fixnum arguments
```

```
(dotimes (x 20)  
  (declare (fixnum x))  
  ...) ⇒ will cause fixnum + routine to be called to increment x
```

---

### NOTE

If you declare something to be a certain type, make sure that is always that type! If you contradict declarations there is no guarantee of safety or system integrity.

---

A type of functional transformation that does not depend on declarations is the transformation of calls to certain functions with keyword parameters, to calls to functions with positional parameters. This bypasses the relatively expensive operation of parsing the keywords at execution time. Of course, this is only possible when the values of the keyword parameters are constants.

Consider the function `member`. It takes three keyword parameters: `:test` (defaults to `'eql`), `:key` (defaults to `'identity`), and `:test-not` (defaults to `nil`). Most of the calls to `member` will be of the form:

```
(member i l)
```

The preprocessor converts this to a call to an internal function that takes three positional parameters: the item being checked for membership, the sequence being checked, and a flag that indicates the status of the `:test-not` keyword. So the above call to `member` would get translated to

```
(member_eq1 i l nil)
```

As would this call:

```
(member i l :test 'eql)
```

## Unsafe Functional Transformations

When the quality `extn:system-lisp` is set to `t`, then the preprocessor will make “unsafe” functional transformations. These are functional transformations that make certain assumptions (regardless of declarations) that may in some cases, turn out to be false. For instance, if you have turned on `extn:system-lisp` with

```
(proclaim '(extn:system-lisp t))
```

then the preprocessor will not generate tests for `nil` arguments to `car`, `cdr`, and other list manipulating functions. If you then call one of these functions with a `nil` argument, the results are unpredictable and possibly hazardous to the state of your Lisp system. The optimizations that will be made when `extn:system-lisp` is on are listed in the “Extensions” section. You should enable this level of optimization only after making sure that the optimizations will not “break” your code.

## Inline Coding

Inline coding is a particular type of functional transformation where the preprocessor inserts the actual code of a called function instead of the code to perform a call to that function. This avoids the execution time overhead of a function call. You can control inline coding of function calls with the `inline` declaration specifier (see the “Types” chapter for details).

## Safety

If `safety` is **proclaimed** to be greater than 0 (1 is the default), the compiler generates code that checks the number of arguments passed to user-defined functions. If `safety` is 0, these checks are not produced. These checks are also not produced when `speed` is 3, regardless of the value of `safety`.

## Extensions

For greater control of optimizations, Hewlett-Packard has added several qualities that are valid for declarations and/or proclamations.

### Compiling vs. Interpreting

There are separate speed qualities for interpreting and compiling. The regular quality `speed` is used when compiling functions with `compile-file`. The quality `extn:eval-speed` is used when the preprocessor is called from `eval` (i.e. the interpreter). This quality can only be proclaimed, not declared. The default for `extn:eval-speed` is 0 (no optimizations are performed). To change it so the preprocessor will optimize interpreted code, execute a form like

```
(proclaim '(optimize (extn:eval-speed 1)))
```

Note that when you compile a function with `compile` (instead of `compile-file`), you are compiling an already preprocessed version of the function, so the optimization of that compiled function is controlled by whatever `extn:eval-speed` was when the `defun` was evaluated. If you specify the optional definition argument to `compile` or use `compile-file` and `load` to define a compiled function, the optimizations are controlled by the declared `speed` quality.

### Closure Optimizations

Closures (function objects) can be upwardly or downwardly mobile. An upwardly mobile closure is one that is used after the form that established the lexical environment in which the closure was created (such as a `let` or `defun`) terminates. A downwardly mobile closure is a closure that is used only while the lexical environment in which it was created still exists.

Here is an example of an upwardly mobile closure.


```
(defun make-fun (f perm-arg)
  #'(lambda (x) (funcall f perm-arg x)))

(setq a-function (make-fun 'nth 0))
(funcall a-function '(a b c)) ⇒ A
```

Here is an example of a downwardly mobile closure.

```
(defun weird-nth-applier (arg1 arg2)
  (let ((fun1 #'(lambda (x) (nth x arg1))))
    (funcall fun1 arg2)))

(weird-nth-applier '(a b c d) 2) ⇒ C
```




HP has added the capability of optimizing downwardly mobile closures. This facility is accessed with the `extn:upward-closures` declaration specifier which may have a value of `nil` or `T`. If `extn:upward-closures` is `T` (the default), then you may use both upward and downward closures. If `extn:upward-closures` is `nil` then the preprocessor assumes that closures are only downwardly mobile, and takes steps to optimize them. The `upward-declarations` specifier can be declared or proclaimed. To optimize `weird-nth-applier`, you could rewrite it as

```
(defun weird-nth-applier (arg1 arg2)
  (declare (extn:upward-closures nil))
  (let ((fun1 #'(lambda (x) (nth x arg1))))
    (funcall fun1 arg2)))
```

### Unsafe Functional Transformations

The quality `extn:system-lisp` has been added to control “unsafe” functional transformations (described above). Valid values for `extn:system-lisp` are `T` or `nil`. To enable unsafe optimizations use

```
(proclaim '(extn:system-lisp t))
or
(declare (extn:system-lisp t))
```




This feature is primarily intended for system implementors. Its effects are:

- Speed 3 optimizations are enabled.
- `car`, `cdr`, `first`, and `rest` are inline with no check for `nil`.
- Certain math functions are coded inline and assume fixnum arguments and results unless declarations are made to the contrary. These functions are: `+` `-` `*` `/` `1+` `1-` `<` `=` `>` `<=` `>=` `/=` `max` `min` `abs` `mod` `rem` `oddp` `evenp` `zerop` `plusp` `minusp` `logand` `logior` `logxor` `logeqv` `lognand` `lognor` `logandc1` `logandc2` `logorc1` `logorc2` `logtest`.

### Optimizing Wisely

The previous section described how HP’s Lisp system makes optimizations to your Lisp programs. This section covers how to responsibly use the power of these optimizations without “shooting yourself in the foot”.

#### Caveats



The benefits of optimization are not without dangers. As discussed above, when optimizing to a certain level, the preprocessor makes assumptions about your code, and it is **your** job to make sure that these assumptions are correct.



One of the biggest bottlenecks in the Lisp system is the overhead of checking parameters to system functions. At run time the system must check the type of each parameter, so that it can let you know if you've passed it something illegal. This takes a lot of time. As you increase the level of optimization and add type declarations, these checks are relaxed, so your code runs faster. However, if you pass a system function a parameter that is the wrong type, the results are unpredictable and sometimes damaging to the stability of the Lisp system. One way to get around this double-edged sword is to do your own parameter checking at strategic points in your program. This technique will be demonstrated shortly.

Another point about system function parameter checking that you need to be aware of is that calls to most system functions are checked for the correct number of parameters by the preprocessor (not when the functions are actually called). This implies that even if you are not optimizing and you invoke a function with `funcall` or `apply`, the system will not check for the correct number of parameters.

Optimizations also make debugging more difficult. Since the preprocessor may change your code before it is interpreted if `extn:eval-speed` is not 0, when you're tracing the code you may not even recognize it. Error messages will be less meaningful because errors will be caught at a lower level (if they're caught at all).

### **Strategy**

These warnings are not here to discourage you from optimizing your code, but to give you an idea of what can go wrong, so you can avoid problems. We suggest that you take the following approach:

- Develop and debug your program without optimizations.
- Determine where it is safe to turn on optimizations.
- Turn on the optimizations in the appropriate places and retest your program.

### **Benefits**

Depending on the nature of your code, making useful declarations and turning on optimizations will make your programs run many times faster than an unoptimized version. Thus it is well worth the effort to optimize carefully.

---

## Using Heap Wisely

Besides speed, another performance measure of a Lisp program is how much heap it allocates while running. (Allocating heap is often called **consing**.) When writing Lisp code, you should be aware of how much your program conses, and take steps to minimize it. When optimizing the amount of consing a program does, you usually trade safety and generality for more efficient use of heap and faster performance (since it takes time to allocate space).

## Destructive Functions

Consider the Common Lisp functions that come in destructive and nondestructive versions: **nreverse** and **reverse**, **nsubstitute** and **substitute**, and so on. (The “n” is a mnemonic for “iN-place”) The destructive functions do not guarantee that the argument they are passed will be unmodified. Instead of consing up all new space for the result, they use the space occupied by the argument. This yields significant performance benefits, but can surprise the unwary.

In this example, the function **reverse-pair**, is **supposed** to return a list whose elements are its argument (a sequence), and the reverse of the argument.

```
(defun reverse-pair (s)
  (let ((rs (nreverse s)))
    (list s rs)
  )
)

(reverse-pair "abcde") ⇒ ("edcba" "edcba")
(reverse-pair '(1 2 3)) ⇒ ((1) (3 2 1))
```

What happened? As soon as **s** is passed to **nreverse**, you can no longer depend on **s** to retain its original value. The second call to **reverse-pair** demonstrates that you also cannot depend on **s** being **(nreverse s)** after **nreverse** returns.

Bugs caused by improper uses of destructive functions (whether user or system defined) are usually more subtle and insidious than the one in **reverse-pair**. When you use or write a utility function, think carefully about whether it is, or should be destructive.

## Shared List Structure

Joe Cobol laughed heartily when he saw the bug in the above definition of `reverse-pair`. "I can fix that easy," he said. Here is Joe's "fixed" definition of `reverse-pair`.

```
(defun reverse-pair (s)
  (let ((save-s s)
        (rs (nreverse s)))
    (list save-s rs)
  )
)

(reverse-pair "abcde") ⇒ ("edcba" "edcba")
(reverse-pair '(1 2 3)) ⇒ ((1) (3 2 1))
```

Joe's program demonstrates another common source of bugs in Lisp programs: unexpectedly shared data. When `save-s` is set to `s`, it is referencing the same piece of data as `s`. When the list referenced by `s` is changed, the change is also reflected in the value of `save-s`. This kind of sharing bug can occur with any Lisp data type **except** numbers, characters, and functions, which are considered to be **immutable**.

Here are some more examples of bugs caused by shared data.

```
(setq array-of-strings
      (make-array 3 :initial-element (make-string 3 :initial-element #\Space)))

(setf (schar (aref array-of-strings 0) 2) #\X)
;; All the elements reference the same string
array-of-strings ⇒ #(" X" " X" " X")

(defun make-rule (p c)
  (let ((template '((premise) (conclusion))))
    ;; Insert p after premise
    (rplacd (car template) (cons p nil))
    ;; Insert c after conclusion
    (rplacd (cadr template) (cons c nil))
    template
  )
)

(setf rule1 (make-rule '(rain) '(turn-on wipers)))
rule1 ⇒ ((PREMISE (RAIN)) (CONCLUSION (TURN-ON WIPERS)))
(setf rule2 (make-rule '(sunshine) '(put-on shades)))
rule2 ⇒ ((PREMISE (SUNSHINE)) (CONCLUSION (PUT-ON SHADES)))
rule1 ⇒ ((PREMISE (SUNSHINE)) (CONCLUSION (PUT-ON SHADES)))
```

The bugs in both of these examples result from data not being allocated the way the programmer expected. In the first one, they expected the `make-string` to be evaluated once for each element of the array; in fact it is evaluated only once, and the result assigned to each element of the array.

The `make-rule` example fails because of a misunderstanding of how constants are allocated. The space for constants is allocated when the constant expression is read (in this case, when the function `make-rule` is defined). Consequently, each call to `make-rule` does not begin with a fresh template. To fix this bug, replace the constant `'((premise) (conclusion))` with `(list (list 'premise) (list 'conclusion))` or better yet, redefine `make-rule` as

```
(defun make-rule (p c)
  '((premise .p) (conclusion .c))
)
```

The method used to allocate constants can be used to reduce consing in certain situations. For instance, if a programmer knew that a rule constructed by `make-rule` was only needed until the next call to `make-rule`, then the first definition would be adequate. In fact, it would be preferable, since the space for the rule template would only be allocated once.

---

## Creating Public Functions

If you are writing functions that will be used by other programmers, you need to include error checking. Here are some general guidelines for doing so.

1. Check parameters with `check-type`, `assert`, `ccase`, `ctypecase`, `ecase`, or `etypecase` in all user visible functions. Make all tests before the parameters are used in a computation that might result in an error. These constructs produce continuable errors so that the debugger can provide friendly feedback to the user. There should be very few explicit calls to `error` or `error` if these macros are used correctly. Always try to catch the error as soon as possible and make it a continuable error rather than postpone the check and give the user a noncontinuable error. Check parameters before they are passed to lower level routines where it will not be as obvious what went wrong. Rest parameters are simply a list so you will need to check the elements of the list individually as you process them.
2. Declare all local variables (in `lets`, `dos`, etc). Do not declare parameters in user visible functions. Parameters in support functions should be declared. Not every local variable or parameter can be declared in a way that will allow the preprocessor to optimize. For example, a variable which may contain a string or `nil` can be declared with `or`, but the preprocessor will not be able to do any optimizing based on this declaration. Use `the` to type-cast the result of an expression when possible. For example, use


```
(the symbol (car foo))
```

when `foo` is known to be a list containing only symbols.

The interpreter can take advantage of this information by checking whether the expression is of the required type. The compiler can use this information to produce better code.

You can write functions that are optimized by the preprocessor without giving up error checking by doing the error check(s) yourself. For instance, suppose you wanted to write a function to call a function `doittoit` with successive elements of a simple vector. One way to do it would be

```
(defun mapv (a)
  (declare (optimize (speed 1))) ; Make it safe
  (dotimes (i (length a))
    (doittoit (svref a i))
  )
)
```




With this definition, if someone calls `mapv` with something other than a vector, a system function (either `length` or `svref`) will find the error. This is not exactly user friendly, but at least nothing disastrous occurred because of the bad parameter. Notice however, that the preprocessor will not be doing much to optimize this code since the speed level is set to 1. Every call to `svref` will have the overhead of checking the type of `a`. We can improve upon the definition of `mapv` in two ways by doing our own error checking.

```
(defun mapv (a)
  (declare (optimize (speed 3)))           ; Make it fast
  (check-type a simple-vector "a simple-vector") ; But safe
  (dotimes (i (length a))
    (doitait (svref (the simple-vector a) i))
  )
)
```

Here we turn the preprocessor loose to do its best, but check for a bad parameter ourselves. We also only do it once, instead of each time through the loop. If a user does call `mapv` with something other than a simple vector, they will get a friendly message that will allow them to continue the error with a new value.

Some things to note about this example:

- 
- The function `doitait` should either work with arguments of any type, or include its own error checking, preferably the former.
  - It is more common to set the optimization level with a `proclaim` that covers all the functions defined in a particular file to avoid repeating the `declare` in every function. There is nothing wrong with overriding that in a particular function.

# Notes



# Types and Declarations

---

## Introduction

One powerful feature of Lisp is its typeless variables. Any variable (unless intentionally restricted) can have any Lisp data object as its value. Lisp **data**, however, does have type. So while you need not worry about assigning to an incorrectly typed variable, you do have to make sure that functions are called with the correct kinds of arguments. For instance, calling `car` with any argument that is not `nil` or a cons is an error.

Although Common Lisp variables are typeless by default, it is possible to declare a variable to be a certain type. Declarations are entirely optional, and their addition or removal will not affect the correctness of a program (except for a `special` declaration, which controls the scope and extent of a variable). Declarations in Common Lisp:

- Increase efficiency of compiled code
- Enable the interpreter to perform additional type-checking
- Enhance program documentation

Common Lisp types are different from the types found in conventional languages. Data types are possibly infinite sets of Lisp data objects, and one should think in terms of an object **belonging** to a type rather than **being** a type. The possible data types form a hierarchy, where a data object that belongs to one type (`integer` for instance) also belongs to any types that superset that type (like `number`).

---

### NOTE

This chapter presents some fundamental information on Common Lisp data types and their use in the Lisp workstation. Types are discussed in greater detail in Chapters 2, 4, and 9 of Steele's *Common Lisp*.

---



---


## Available Types

The following list shows some of the basic data types available in Common Lisp. Since these basic types can be combined or restricted to form new types, this is not a complete list of every possible type. (Such a list would be difficult to come by, since users can define their own types.)

Symbol	Symbols are abstract data objects that are notated by their name, which consists of a series of characters. Each symbol has a home package, a value cell, a function cell, and a property list which serves as a structure for storing additional information about that symbol. Functions and variables in Lisp are named by symbols, so Lisp must provide a means of accessing a symbol with its name.
List	A list is an ordered sequence of data objects that is represented by linked cells called conses. Each cons cell holds a pointer to the next object in the list (the <code>car</code> ), and a pointer to the remainder of the list (the <code>cdr</code> ). The special symbol <code>nil</code> stands for the empty list. Items in a list may themselves be lists.
Number	There are several types for representing numbers including an integer type and a double precision floating point representation.
Character	This data type provides a way of storing the graphic symbols commonly used in writing. This includes letters, punctuation, and a few non-printing characters that have special meaning (such as <code>#\Newline</code> ). Collections of one or more characters can be stored in arrays.
Array	The concept of an array in Common Lisp is the same as in other programming languages. An array is a dimensioned collection of data whose elements can be referenced by a series of integers (one integer per dimension of the array). General Lisp arrays can store any Lisp data object, but arrays can also be restricted to hold only elements of a given type.
Vector	A vector is a one-dimensional array.
String	A string is a vector whose elements are of type <code>string-char</code> .
Structure	A structure is like a Pascal record or a C structure. A user may define a data type whose members are data objects with named components. These data objects are structures.
Function	Functions take arguments and return results, while possibly causing some side-effects. Since variables can be bound to functions, functions are considered a class of data.
Instances	This is a type that Hewlett-Packard has added to its implementation of Common Lisp to facilitate object-oriented programming. See the chapter "Object-Oriented Programming" for details.

---

## Type Specifiers




To specify the set of data objects that constitute a type you wish to use, Common Lisp provides **type specifiers** that may be used in places appropriate for a type definition (like the declarations described later in this chapter).

Type specifiers can be symbols or lists. Common Lisp defines some standard type symbols. These are listed on p. 43 of Steele's *Common Lisp*. In addition to these, the symbol `instance` is a valid type specifier symbol in Hewlett-Packard's Common Lisp.

The first element of a **type specifier list** is a symbol, and the remainder of the list (if present) is further information about the type being specified. Common Lisp predefines several symbols that may be used at the beginning of a type specifier list, as well as allowing you to create your own with the `deftype` macro. The remainder of a type specifier list consists of parameters that determine exactly what data objects are members of the type described by that type specifier list. For example,

```
(integer 0 7) ; the integers from 0 to 7
```



If you don't want to designate a particular parameter in a specifier list, then put a `*` in the position for that parameter. Here's a list that specifies the type consisting of all vectors of length 4:

```
(vector * 4)
```

In this case, the `*` takes the place of the parameter that, if given, indicates the type of the vector elements. Since it is not specified in this example, the type includes any vectors of length 4. Any `*`'s that appear at the end of the list can be discarded. For instance, the type specifier list

```
(vector integer *)
```

denotes the set of integer vectors of any length, and is abbreviated with

```
(vector integer)
```

## Subset Type Specifiers

There is a certain amount of overhead required for storing Lisp data objects. Allowing a variable to store any possible data object (the default condition) means that the overhead for that variable will be the maximum possible. For this reason, Common Lisp provides type specifier lists that specify specializations of other data types. Note that using these specifiers in a declaration only allows a Common Lisp implementation to use a more efficient representation, it does not require it.

The following text describes in detail a few of the symbols that Common Lisp defines for use in this context. The remaining symbols are listed; for details on the precise semantics of these, see Chapter 4 of Steele.

### Array

(array *element-type dimensions*)

A type specifier list whose first element is `array` describes a subset of the possible array objects. *Element-type* is a type specifier that indicates what type of elements an array in the type being described is to hold. *Dimensions* is either a non-negative integer that indicates the number of dimensions of the array, or a list of non-negative integers that represent the length of each of the array's dimensions. The length of any of the dimensions can be left unspecified by putting a `*` in place of the appropriate integer. Unlike the `*`'s that appear within the highest level of a type specifier list, these can never be omitted.

### Examples:

```
(array integer 4)           ; Four-dimensional integer arrays
(array list (2 * *))       ; Three-dimensional arrays of lists
                           ;   whose first dimension has length 2
(array list *)             ; Arrays of lists
(array list)               ; Arrays of lists
(array * (2 2 2))         ; All 2 by 2 by 2 arrays
```


### Integer Subranges

You can specify a type that is a subrange of the integers with a type specifier list that starts with `integer`. This takes the form of

(integer *lower-bound upper-bound*)

*Lower-bound* and *upper-bound* must each be either an integer, a list of a single integer, or `*`. If a bound is an integer, the type includes that integer; if it is a list of an integer, that integer is not included in the type. If `*` is given as a bound, then that bound does not exist, and the range ends at plus or minus infinity.


## Examples:



```
(integer * (0))           ; All negative integers
(integer (0) *)          ; All positive integers
(integer 0 *)            ; Non-negative integers
(integer -1 3)           ; The set {-1, 0, 1, 2, 3}
(integer -1 (3))         ; The set {-1, 0, 1, 2}
(integer (-1) (3))       ; The set {0, 1, 2}
```

## Other Subset Type Symbols

The following is a list of the additional symbols that are available for specifying types that are basically subsets of other types. Most of them are self-explanatory, but if necessary, see Steele for details on their exact meaning.



```
(vector element-type length)
(simple-vector size)
(function (arg1-type arg2-type...) value-type)
(values value1-type value2-type...)
(mod n)
(signed-byte s)
(unsigned-byte s)
(float lower-bound upper-bound)
(short-float lower-bound upper-bound)
(single-float lower-bound upper-bound)
(double-float lower-bound upper-bound)
(long-float lower-bound upper-bound)
(string length)
(simple-string length)
```



Type lists that have a single length or size parameter include only objects that have exactly that length or size.

Use of the type specifying symbol `function` is restricted. It may only be used in declarations; it cannot be used for type-checking.

The symbol `values` is also restricted. It can only be used as the value type specifier in a function type specifier, or in a the special form.

## Predicate Type Lists

You can specify a type to include any data object that satisfies a given one-argument predicate. This is done with a type specifier list that begins with the symbol `satisfies` and whose one other element is the name of the predicate to be satisfied. For instance, if we define `our-even-p` as

```
(defun our-even-p (x)
  (and (integerp x) (evenp x)))
```

then

```
(satisfies our-evenp)
```

denotes the type that is the set of all even integers. Note that we could've run into trouble if we had just used `evenp` in the `satisfies` form since it is an error to call `evenp` with any object other than an integer. The `and` in `our-even-p` prevents `evenp` from being called unless `x` is an integer. You should make sure that any predicate function you use in a `satisfies` type specifier list has no side effects.

### Examples:

```
(satisfies symbolp)           ; All symbols
(satisfies functionp)        ; All functions
```

There are a few other type specifiers that allow combinations of types similar to those possible with `satisfies`.

A list (`member object1 object2 ...`) denotes the type consisting of the specified objects. For instance,

```
(declare (type (member 'orange 'lemon 'grapefruit) fruit1 fruit2))
```

says that `fruit1` and `fruit2` will only take on one of the three values `orange`, `lemon`, or `grapefruit`.

(*not type*) specifies the set of all objects not in *type*.

(*and type1 type2 . . .*) specifies the intersection of the types listed.

(*or type1 type2 . . .*) specifies the union of the types listed.

The *and* and *or* type specifying symbols work similarly to their equivalent logical operators. When an *and* type specifier is processed, the object being tested is tested against each of the given types from left to right. As soon as the object fails one of the tests, processing stops and *nil* is returned. This allows you to create a “sieve” of types so that if one of the types is a *satisfies* specifier whose predicate only takes arguments of a certain type (like *evenp*), the *and* can be constructed to eliminate improper arguments before the predicate is called.

For example, if you wanted to specify a type consisting of all even integers, you could use

```
(and integer (satisfies evenp))
```

This specification prevents calling *evenp* with a non-integer argument, since if the object being tested is not an integer, the test will fail before getting to the *satisfies* specifier.

An *or* type specifier list works analogously: the object being tested is checked against the given types from left to right until it is found to belong to one, or runs out of types to check it against.

## Defining Type Symbols

The macros *defstruct*, *deftype*, and *define-type* let you define new type symbols to use in type specifiers. The name of a structure type defined by *defstruct* will be a valid type specifier symbol (unless it was defined with the *:type* option). The macro *deftype*, is a macro-like facility that allows you to create your own type specifying symbols that may or may not take arguments. The macro *define-type* is for defining instance types for object-oriented programming.

A call to *deftype* has the following form:

```
(deftype type-name lambda-list {declaration [doc-string]* {form}*) Macro
```

*Type-name* is the type specifier symbol you are defining; *lambda-list* is a lambda list that describes the arguments that may be given with *type-name* in a type specifier list; any *forms* given are the body of the type definition. *deftype* returns *type-name*.

When a symbol defined by *deftype* occurs in a type specifier list, any argument forms are bound to the corresponding parameters in the lambda list of the definition. The arguments are not evaluated. The forms that make up the body are then evaluated as an implicit *progn*, and the value of the last form evaluated becomes the type specifier that the original specifier list represents.

### Example:

```
; Define a type that consists of all integer multiples of 16
(defun hexdivp (n)
  (= 0 (rem n 16)))

(deftype multiple16 ()
  '(and integer (satisfies hexdivp)))
```

After this definition form is evaluated, if you use the type specifier (`multiple16`) it stands for the type `(and integer (satisfies hexdivp))`.

The lambda list in a `deftype` form is completely general, so you can use `&optional` and `&rest` markers. However, if no `initform` is given for an `&optional` parameter, the default value is `*`, instead of `nil`. `&optional` parameters are used in the following example in which we define a type specifier symbol (`cubic-array`) that lets us specify three dimensional arrays whose dimensions all have the same length. The definition of `cubic-array` allows a programmer to optionally indicate the type of the elements and/or the size of the dimensions.

### Example:

```
(deftype cubic-array (&optional elt-type dim-size)
  '(and (array ,elt-type (,dim-size ,dim-size ,dim-size))
        (satisfies equaldimp)))

(defun equaldimp (a) ;; Returns t if sizes of a's dimensions are =
  (or (< (array-rank a) 2)
      (apply #'= (array-dimensions a))))
```

Here are a few examples of how some uses of `cubic-array` would be expanded.

`(cubic-array float 5) ≡ (and (array float (5 5 5)) (satisfies equaldimp))`

`(cubic-array * 9) ≡ (and (array * (9 9 9)) (satisfies equaldimp))`

`(cubic-array integer *) ≡ (and (array integer (* * *))  
                                  (satisfies equaldimp))`

`(cubic-array integer) ≡ (and (array integer (* * *))  
                                  (satisfies equaldimp))`

`(cubic-array) ≡ (and (array * (* * *)) (satisfies equaldimp))`

`cubic-array ≡ (and (array * (* * *)) (satisfies equaldimp))`

Notice from the last example that when no arguments are specified to `cubic-array`, you can use just the symbol itself; there is no need to enclose it in parentheses (although there is nothing wrong with doing so).



---

## Declarations

One context in which type specifiers can be used is declarations. Declarations are a way of specifying additional information about the bindings of variables and certain other aspects of your Lisp program. There are three basic types of declarations:

1. Those that deal with variable bindings. These are primarily concerned with the typing of variables.
2. Those that deal with things other than variable bindings. This category includes declarations that change how the compiler treats the affected portion of code.
3. Special declarations. These fall into both of the above classes and are of a global nature. Special declarations are the only type of declarations that affect the meaning of a program.

Note that declarations in Common Lisp can affect things other than variable data types. For example, there are declarations that change scoping, the environment of a function, or the way the compiler treats function calls.

The nature of declarations in Common Lisp differs significantly from the nature of declarations in other high level languages. In Common Lisp, declaring a variable to be a certain type **does not force the object stored in that variable to be that type**. A declaration is your “promise” to the language processors that a variable will always be the type you have declared it to be. The following example illustrates this.

```
(let (v)                                ;; This code will not work.
  (declare (simple-vector v))           ;; Do not try to run it.
  (dotimes (i 50)
    (setf (svref v i) 1)))
```

Here, `v` has been declared to be a simple vector. However, the value stored in `v` is `nil`. Depending on the current level of optimization, the preprocessor may not check to see if `v` is really a simple vector (since you have said that it is in your declaration) before it tries to do the `svref`. Using `svref` on `nil` causes unpredictable results. The correct version of this example is

```
(let ((v (make-array 50)))              ;; This code will work.
  (declare (simple-vector v))
  (dotimes (i 50)
    (setf (svref v i) 1)))
```

## Symbols for Declaring

Declarations can be made with three different symbols: `declare` (a special form), `locally` (a special form), and `proclaim` (a function). Which one you use depends on where you want your declarations to take effect and what type of declaration you are making.

- `declare` is used to imbed any type of declaration in executable code.
- `locally` makes declarations that are local to the form in which they occur. It cannot be used to make declarations that affect variable bindings.
- `proclaim` is for making global declarations.

### The `declare` Special Form

To imbed declarations within code (such as within a function definition) use the `declare` special form, which has the syntax

```
(declare {declare-specification}*) Special Form
```

*Declare-specification* is a list whose first element is one of several symbols (to be described shortly) that indicate what type of declaration is being made.

`declare` forms can only occur before the body of certain special forms and macros. For example,

```
(defun inker (i)
  (declare (integer i))
  (+ i 1))
```

Note that all declarations within a special form must precede any other statements in the body of that form. Declarations affecting variable bindings apply only to the bindings made by the form in which they appear, but declarations not concerned with bindings affect any code within the form. For instance, `(declare (type integer x))` only affects bindings of `x` made by the form in which it appears, while `(declare (inline foo))` affects all calls to `foo` textually contained within the form. Here are the names of all the special forms and macros that permit declarations:

<code>defmacro</code>	<code>dotimes</code>
<code>defsetf</code>	<code>flet</code>
<code>deftype</code>	<code>labels</code>
<code>defun</code>	<code>let</code>
<code>do*</code>	<code>let*</code>
<code>do-all-symbols</code>	<code>locally</code>
<code>do-external-symbols</code>	<code>macrolet</code>
<code>do-symbols</code>	<code>multiple-value-bind</code>
<code>do</code>	<code>prog</code>
<code>dolist</code>	<code>prog*</code>
<code>lambda</code>	<code>extn:define-method</code>

### The locally Special Form

The `locally` special form makes declarations that apply only to the forms inside it. Since `locally` does not establish any bindings, declarations that affect variable bindings (like `type`, etc.) are not meaningful in a `locally` form. However, a `locally` can affect references to variables, so you can declare variable references to be special within a `locally` form.

```
(locally {declaration}* {form}*)
```

*Special Form*

All listed *declarations* apply to the forms within the `locally`. `locally` returns whatever the last *form* returned. If there are no forms following the declarations, then *nil* is returned.

```
(defun two-calls (l)
  (declare (notinline cdr))
  (cons (locally (declare (inline cdr))
                (car (cdr l)))
        (cdr l)
        ))
```

There are two calls to `cdr` in this example. The first call will be compiled in-line. The second call to `cdr` will not be since it is outside the `locally` declaration requesting that calls to `cdr` be done in-line.

Here's an example of a special declaration made in a `locally`.

```
(setq y 67)

(let ((y 3))
  (+ y
     (locally (declare (special y))
              (setq y 8))
     )
  ) ⇒ 11

y ⇒ 8
```

This particular `locally` form declares that any occurrences of `y` that appear within it refer to the current special binding of `y`.

## The `proclaim` Function

To make a global declaration, use the `proclaim` function.

```
(proclaim declare-specification)
```

*Function*

Because `proclaim` is a function, its argument is always evaluated. This lets you compute a declaration on the fly and means you sometimes must quote the specifications for them to work. Variable names used in *declare-specification* refer to the dynamic values of the variables. Function names refer to their global function definition. Declarations made by `proclaim` are global, but they may be overridden by a local declaration. For instance, the global declaration

```
(proclaim '(notinline fac))
```

could be overridden in a function `zoo`.

```
(defun zoo (i j)
  (declare (inline fac))
  (+ j (fac i)))
```

Notice that the quote in the first example (the `proclaim`) is not used in the `declare` special form since its argument is not evaluated.

---

### NOTE

Use `proclaim` with caution. Be certain you understand its effects before using it.

---

## Symbols for Specifying Declarations

This section describes the symbols that can be used as the first element of a declaration specification (either in a `declare` special form or `proclaim` function call).

### **special**

```
(special var1 var2 ...)
```

`special` specifies that all of the variables in the list are special. For information on what it means for a variable to be special, see the “Scope and Extent” chapter. All affected variable bindings are made dynamic and references to the named variables refer to that dynamic binding.

A `special` declaration is not pervasive. Inner bindings of a variable override a `special` declaration and must be redeclared to be special (if desired). For example,

```
(defun kung (a b)
  (declare (special a))
  (let ((a 6))
    (+ a b)))
```

In `(+ a b)`, `a` refers to the binding established by the `let` and not the special binding in `kung`. If `kung` was rewritten as

```
(defun kung (a b)
  (declare (special a))
  (let ((a 6) (declare (special a))
    (+ a b)))
```

then the bindings of, and references to, `a` are special throughout `kung`.

As an exception to the above, if a variable is **proclaimed** to be special, then that declaration applies to all bindings and references of that variable.

```
(proclaim '(special a))

(defun kung (a b)
  (let ((a 6))
    (+ a b)))
```

In this case, both the binding of `a` established by the function `kung`, and the binding established by the `let` form are special. Using `proclaim` to declare a special variable is not advised. The macros `defvar` and `defparameter` are the accepted ways of defining globally special variables.

### **type**

`(type type var1 var2 ...)`

A **type** declaration specifier affects only variable bindings. It declares that the variables in the list will only take values of type *type*. For example,

`(type (integer 1 1000) i j k)`

### **type**

`(type var1 var2 ...)`

This is equivalent to `(type type var1 var2 ...)`, but *type* must be one of the standard type specifiers. It cannot be a user-defined type or a type specifier more complex than a single symbol. For example,

`(integer i j k)`

### **ftype**

`(ftype type function-name1 function-name2 ...)`

A declaration specifier like this says that the named functions are of the functional type *type*. For example,

`(ftype (function (integer array) float) fetch getter)`

This says that `fetch` and `getter` are the names of functions that take two arguments — an integer and an array — and return a float.

### **function**

`(function name arg-type-list result-type1 result-type2 ...)`

This is equivalent to

`(ftype (function arg-type-list result-type1 result-type2 ...) name)`

For example,

```
(function fetch (integer array) float)
```

**inline**

```
(inline function1 function2 ...)
```

An **inline** declaration specifier tells the compiler that you want the named functions expanded inline. This does not force the compiler to do so, it only indicates that it is desirable. Inline expansion is the insertion of a function's code into the calling routine, thus avoiding the overhead of a function call.

**notinline**

```
(notinline function1 function2 ...)
```

This declaration specifier tells the compiler that you do not want calls to the named functions expanded inline. The compiler must follow this advice.

**ignore**

```
(ignore var1 var2 ...)
```

The **ignore** declaration specifier specifies that the bindings of the named variables are not used in the scope of this declaration.

## optimize

```
(optimize (quality1 value1) (quality2 value2) ...)
```

The `optimize` declaration specifier directs the preprocessor to optimize various qualities to a specified level. *qualityn* is a symbol. The following qualities are recognized in Hewlett-Packard Common Lisp: `speed`, `space`, `safety`, `compilation-speed`, and `extn:eval-speed`. The qualities `space`, and `compilation-speed` currently have no effect. The `extn:eval-speed` quality may only be proclaimed, not declared.

*Valuen* is a non-negative integer between 0 and 3. 0 says that the corresponding quality is totally unimportant, and 3 specifies the highest level of importance. *quality* is equivalent to `(quality 3)`. See the “Programming Tips” chapter for information on what effect the various qualities have on optimization.

Example:

```
(defun danger-danger (x y list)
  (declare (optimize (speed 3) (safety 0)))
  (+ x (nth y list)))
```

## declaration

```
(declaration name1 name2 ...)
```

The `declaration` type specifier tells the compiler that the given names are valid symbols for use as the first element of a declaration specifier. It can be used only with a `proclaim` form, not a `declare`. The named symbols can then be used as declaration specifiers for either `proclaim` or `declare`. Use this so the compiler won't issue warnings about declarations that were intended for use by another program processor or language implementation.

## extn:system-lisp

```
(extn:system-lisp {T | nil})
```

An HP extension, the `system-lisp` declaration specifier toggles whether the preprocessor makes “unsafe” optimizations when compiling code. Declaring or proclaiming `system-lisp` to be `non-nil` turns on the unsafe optimizations; `nil` turns them off. For more information, see the “Programming Tips” chapter.



### **extn:upward-closures**

```
(extn:upward-closures {T | nil})
```

The `upward-closures` specifier is an HP extension that allows the language preprocessor to optimize code that uses only downward closures. If declared or proclaimed to be `nil` (it defaults to `T`), then the preprocessor assumes that all closures in the affected code are downward. For more information on this specifier, and upward and downward closures, see the “Programming Tips” chapter.

### **extn:name**

```
(extn:name symbol)
```

This is an HP extension that can only be used in a `declare` inside a lambda expression. It allows you to name a lambda expression. This is sometimes useful for debugging since the name of the lambda expression will be available to debugging tools like the stack browser.

```
(setf thor #'(lambda () (declare (extn:name thor)) (+ 1 2)))
```

### **extn:version**

```
(extn:version string)
```

This is an HP extension that can only be used in a `proclaim`, and only has effect in a file being compiled by `compile-file`. It embeds *string* in the resulting binary file. This is useful when using HP-UX SCCS commands such as *what(1)*.

### **warn**

```
(warn {T | nil})
```

This is an HP extension that can only be used in a `proclaim`. Declaring `warn` to be `nil` suppresses all but the most important preprocessor and compiler warning messages; declaring it to be `T` enables the printing of all preprocessor and compiler warnings.

---

## Other Uses of Types

Besides declarations, there are a few other uses for types in Common Lisp, including specifying the type returned by a particular form, checking the type of a variable, and controlling program flow.

### Specifying the Type of a Form

To increase efficiency, it is sometimes desirable to declare the type of value a specific form will return. This is the purpose of the `the` special form.

(`the` *return-value-type* *form*)

*Special Form*

This is your guarantee to the compiler that the form will return a value of the type you have specified. The following two definitions of `fast-and-dirty-car` achieve the same effect, but the second one uses `the` instead of `declare`.

```
(defun fast-and-dirty-car (l) (declare (list l))
  (car l))
```

```
(defun fast-and-dirty-car (l)
  (car (the list l)))
```

Here's an example where `declare` cannot be used, but `the` can.

```
(defun exists (i l)
  ;; Returns i if it is found at top-level in l,
  ;; otherwise, it returns nil
  (loop (ctypecase l
        (cons (if (eq i (car (the cons l)))
                  (return-from exists i)
                  (setq l (cdr (the cons l))))))
        (nil (return-from exists nil))
        )
  )
)
```

In the `cons` part of the `typecase` we already know that `l` is a `cons`. If we declare it as such with `the`, then depending on the current value of `extn:eval-speed` or `speed`, the preprocessor will replace the calls to `car` and `cdr` with inline code that doesn't check for `nil`, thus making the code more efficient.

Here are a few more simple examples of the:

```
(the float (sqrt (abs x)))
```

```
(the symbol name) ; form does not have to be a list
```

```
(the (vector * 6) (aref a 1)) ; a must be an array of vectors of length 6
```

## Checking Types

Common Lisp provides several functions for checking the types of variables. One key thing to remember when type-checking is that the Common Lisp compiler may not be able to use special representations for all declared variables. Thus, it is possible for a variable you have declared to be a certain type not to be that type.

You can obtain the type of a given object with the `type-of` function.

```
(type-of object)
```

*Function*

This returns the type of object. For portability, you should be aware that Common Lisp implementations are only required to return **some** type of which the object is a member. Consequently, `type-of` is not very useful and is best reserved for debugging. Hewlett-Packard has extended the `type-of` function to return the name of the instance type of its argument when that argument is an instance. (See the chapter “Object-Oriented Programming” for details.)

## Type Membership Predicates

The general predicate for testing whether an object is a member of a given type is `typep`.

```
(typep object type)
```

*Function*

The function `typep` returns `t` if `object` is a member of `type`, and `nil` otherwise. `Type` can be any valid type specifier except that it may not contain either of the type specifying symbols `function` or `values`. Remember that objects can be members of more than one type. Example:

```
(typep 46 'integer) ⇒ T
```

```
(typep 46 'number) ⇒ T
```

In addition to `typep`, there are predicates that can test whether an object is a member of a specific type. For instance, the predicate `symbolp` returns `t` if its single argument is a symbol. With the exception of `atom` and `null`, the type checked for is the name of the predicate with the `p` or `-p` on the end omitted. `atom` returns true only if its argument is not a cons, and `null` returns true only if its argument is `nil`.

(null object)	Function
(atom object)	Function
(symbolp object)	Function
(consp object)	Function
(listp object)	Function
(numberp object)	Function
(integerp object)	Function
(rationalp object)	Function
(floatp object)	Function
(complexp object)	Function
(characterp object)	Function
(stringp object)	Function
(bit-vector-p object)	Function
(vectorp object)	Function
(simple-vector-p object)	Function
(simple-string-p object)	Function
(simple-bit-vector-p object)	Function
(arrayp object)	Function
(packagep object)	Function
(functionp object)	Function
(compiled-function-p object)	Function
(commonp object)	Function
(extn:instancep object)	Function

Another means of checking the type of an object is the macro `check-type`. `check-type` is commonly used to check the types of function parameters.

`(check-type place type-specifier &optional type-string)` *Macro*

*Place* must be a variable reference valid for `setf`, and *type-specifier* can be any type specifier. *Type-specifier* is not evaluated, so should not be quoted.

`check-type` uses `typep` to check whether the value in *place* belongs to the type specified by *type-specifier*. If it does not, the error message

```
!!!! Continuable error: The value of place, value, is not type-string  
If continued: prompts for a value to use.
```

is displayed and you can enter a new value.

The optional parameter *type-string* should be a string, and will be printed as the desired type in the error message. If *type-string* is not given the type to be displayed in the message is determined from the type specifier.

### Comparing Types

To check whether one type is a subtype of another, use `subtypep`.

`(subtypep type1 type2)` *Function*

The two type specifiers can be any type specifiers that do not use the type specifying symbols `function` or `values`. Two values are returned by `subtypep`. Both the values are `t` if *type1* was definitely determined to be a subset of *type2*. If the first value returned is `nil` and the second value is `t`, then *type1* is definitely not a subset of *type2*. If both returned values are `nil`, then `subtypep` could not conclusively determine the relationship between the two types.

```
(multiple-value-list (subtypep 'fixnum 'integer))    ⇒ (T T)  
(multiple-value-list (subtypep 'simple-vector 'cons)) ⇒ (NIL T)  
(multiple-value-list (subtypep 'extn:instance 'integer)) ⇒ (NIL NIL)
```

## Program Control With Types

Common Lisp has three macros that alter the control flow of a program based on the type of a given form. The most general of these is `typecase`.

```
(typecase keyform Macro
  (type-1 consequent-1-1 consequent-1-2 ...)
  (type-2 consequent-2-1 ...)
  (type-3 consequent-3-1 ...)
  ...)
```

The `typecase` macro evaluates *keyform*, then compares the type of the value it returned to the types in the `typecase`. The consequents associated with the first type to which the returned value belongs are evaluated. A type selector of `t` or `otherwise` matches the type of any returned value. If none of the cases are matched, `nil` is returned

```
(defun generic-first (thing)
  (typecase thing
    (cons (car (the cons thing)))
    (nil nil)
    (string (char thing 0))
    (simple-vector (svref thing 0))
    (vector (aref thing 0))
  )
)
```

Notice from this example that the types do not have to be mutually exclusive. However, if one of your types is a subset of another, the more specific type should come first. For instance, in `generic-first`, it would not make sense to have `vector` before `simple-vector` because then the `simple-vector` clause could never be selected.

The macros `etypcase` and `ctypcase` serve the same purpose as `typecase` but provide built-in error handling. The syntax is the same, but no `t` or `otherwise` clause is allowed. In `etypcase`, if no clause is satisfied, then an error message is printed. If `generic-first` had been defined with `etypcase` instead of `typecase`, then `(generic-first 'foo)` would produce the message

```
!!!! Error: The key value FOO does not correspond to any of
           the types (VECTOR SIMPLE-VECTOR STRING NIL CONS).
```

The `ctypcase` macro works the same way, but the error is continuable. If you decide to continue, you will be prompted for a new value for the offending variable.

## Type Coercion

`coerce` converts a data object into an equivalent object of another type. Of course, not all coercions are possible: attempting an illegal coercion results in an error. Here are the allowed conversions:

An object that belongs to any sequence type can be coerced to any other sequence type provided that the elements are of the correct type.

Strings of length one can be coerced into characters. `coerce` returns the single character of the string.

```
(coerce "c" 'character) ⇒ #\c
```

Symbols whose print name has length one can be coerced into characters. `coerce` returns the single character of the symbol's print name.

```
(coerce 'x 'character) ⇒ #\X
```

When converting an integer to a character, `coerce` returns the result of calling `int-char` with that integer.

The only provided floating point representation is `double-float`. You can ask for any number to be coerced into `float`, `short-float`, `single-float`, `double-float`, or `long-float`, and it will always be coerced into `double-float`.

Any number can be coerced into a complex number.

Any data object can be coerced into type `t`.

---

## Examples of Declarations

The following is a list of examples of declarations. There are two sections: declarations that can be made at top-level, and declarations to be made within the special forms and macros that allow them.

### Global Declaration Examples

These declarations are made with `proclaim`. They are usually made at top-level, and are in effect anywhere they are not overridden.

```
(defvar *indent-level* 0) ; Make all bindings of *indent-level* special.
                          ; This declaration cannot be overridden.

(proclaim '(notinline foo)) ; Calls to foo should not be made inline

(proclaim '(declaration conjunction ; Allow the symbols conjunction and
          premise) ; premise to be used in declarations

(proclaim (current-optimization-level)) ; Current-optimization-level must
                                          ; return a list that is a valid
                                          ; declaration
```

### Local Declaration Examples

Here are examples of declarations made with the `declare` special form. These can be made anywhere a declaration is allowed (see the list of special forms and macros that allow declarations earlier in this chapter).

```
(declare (simple-string name rank serial-number)) ; Name, rank, and
                                                  ; serial-number are
                                                  ; simple-strings

(declare (type (simple-vector 20) x y)) ; x and y are simple-vectors
                                         ; of length 20

(declare (simple-vector x)) ; x is a simple-vector

(declare (type simple-vector x)) ; x is a simple-vector

(declare (integer a b c)) ; a, b, and c are integers

(declare (type (integer 0 *) a)) ; a is a non-negative integer

(declare (special donut)) ; Make this a special binding
                          ; of donut
```



```
; discombobulate is a function that takes two float arguments, and
; returns an integer.
(declare (ftype (function (float float) integer) discumbobulate))

; Same as previous.
(declare (function discumbobulate (float float) integer))

(declare (inline foo)) ; Request that calls to foo be made inline
(declare (notinline foo)) ; Calls to foo should not be made inline
(declare (optimize (speed 0))) ; Turn off optimizing
```

# Macros

---

## Introduction

There are at least three good reasons for writing Lisp macros:

- Delayed evaluation of arguments
- Reduced call overhead
- Convenient shorthand for commonly used pieces of code

Sometimes a function just won't do. For instance, how successful would you be if you tried to write a function with the functionality of the macro or? Not very, because the forms you need to evaluate inside the function (to see if they return true) would already have been evaluated before they were passed to the function. The arguments to a macro, on the other hand, are passed unevaluated. This gives the macro programmer the choice of when or whether to evaluate arguments.

Most of the overhead of a macro call occurs at preprocess time when the macro call is expanded. Unlike a function call, the execution of a macro call does not require constructing a new activation record to push on the execution stack. Depending on the situation, this means that a macro can be more efficient than a function.

Learning to write Lisp macros is like learning to ride a bike: experience is the best teacher. However it's very difficult to learn without any background information. This chapter is macro training wheels. It gives some fundamental advice about writing macros, and discusses a few simple examples. There is more detailed, but less accessible information about macros in Steele.

## Background

Common Lisp macros provide the ability to execute possibly complex series of forms with a simple macro call. Prior to execution, a macro call is replaced with another form that is computed by the definition of the called macro. This substitution is the essence of macros.

### Expansion Time

It is important to distinguish between macro expansion time and execution time. Expansion time is when a macro call is replaced with the form computed according to the macro's definition. This occurs when the macro call is first processed. For instance, if you define a function `foo` that calls a macro, the macro call is expanded when the `defun` is evaluated. If you then change the definition of the macro, `foo` will still contain the macro expansion computed by the original version of the macro. To make `foo` use the new definition of the macro, you must reevaluate the `defun`.

Since the definition of a macro must be available before the expansion of a call can be computed, macros must be defined before they are used as part of the definition of a function (or otherwise called).

When writing macros, it is often helpful to see the expansion of a call. The functions `macroexpand` and `macroexpand-1` give you that capability.

```
(macroexpand form &optional env)
(macroexpand-1 form &optional env)
```

*Function*  
*Function*

*Form* should be a quoted version of the macro call you wish to see the expansion of. These functions return two values:

- The expanded version of *form* if it is indeed a macro call, otherwise *form* is returned.
- `T` if *form* is a macro call, otherwise `nil`.

The difference between the two functions is that `macroexpand-1` will only expand the outermost macro call, whereas `macroexpand` will keep expanding the form until it is no longer a macro call. Usually, `macroexpand-1` is the most useful for debugging your own macros. You may want to expand some calls to system macros or some of the examples in this chapter just for practice. For example,

```
(pprint (macroexpand-1 '(or (car foo) (or (numberp x) x?) radio))) =>
(LET (#:G9
      (IF (SETQ #:G9 (CAR FOO))
          #:G9
          (IF (SETQ #:G9 (OR (NUMBERP X) X?)) #:G9 RADIO)))
```

## Definition

Macros are defined with `defmacro`. Its syntax is nearly identical to `defun`.

`(defmacro name lambda-list {declaration | doc-string}* {form}*)` *Macro*

There are several extensions to *lambda-list* allowed in a `defmacro`. The use of these extensions is not covered in this chapter, but you should be aware of their existence.

The following additional lambda-list keywords are allowed.

- &body** This is the same as a `&rest`, except that it tells some output-formatting functions that the rest of the form should be treated as a body. You may use only one of either `&body` and `&rest`.
- &whole** The keyword `&whole` followed by a variable specifies that the variable should be bound to the whole macro call form. The `&whole` and variable should be the first things in the lambda-list.
- &environment** The keyword `&environment` followed by a variable specifies that the variable is to be bound to an environment representing the lexical environment in which the macro call is to be interpreted. This is an esoteric option that you may never need. See Steele for more information.

Lambda lists in a `defmacro` can also use a facility known as **destructuring**. Briefly, anywhere in the lambda list where a list is not normally allowed, you can replace a parameter with another lambda list. The argument associated with that parameter in a call is treated as a list whose components are matched with the parameter lambda list.

The body of a `defmacro` consists of the forms to be evaluated at macro expansion time. The value of the last form evaluated is the form that will be substituted for the macro call.

The following very simple macro provides a shorthand way to call a particular function with specific arguments.

```
(defmacro stupid ()  
  '(the-name-of-a-function-thats-much-too-long "Ahab" 3)  
)
```

A call to the macro `stupid` would always be replaced with the form

```
(the-name-of-a-function-thats-much-too-long "Ahab" 3)
```

which would then be evaluated at execution time.

This is not too useful, because there is no use of parameters. Suppose we wanted to make `stupid` a macro that would act as a synonym for `the-name-of-a-function-thats-much-too-long`. This could be defined as

```
(defmacro stupid (arg1 arg2)
  (list 'the-name-of-a-function-thats-much-too-long arg1 arg2)
)
```

Note that we couldn't use a single quoted list because we needed to have the values of the arguments substituted in. We had to use `list` to construct a form to return as the expansion of the macro call. For macros where the form to be constructed is not simple, using `list` in this manner would become tedious. Fortunately, there is an easier way.

### The Backquote Character

The backquote character (`'`) provides a convenient way of constructing a return form within a macro definition. Its use is not restricted to macro definitions, but that is where it is most often used.

The backquote is similar to a quote in that it specifies that the form that follows is to be taken literally. Backquote however, allows the use of commas to insert values into the backquoted form. Within a backquoted form, the following uses of commas are allowed:

`,form` The result of evaluating `form` is inserted in place of `,form`.

`,@form` The result of evaluating `form` should be a list. This resulting list is "spliced" into the backquoted form. e.g. `'(+ ,@(cdr '(2 3 5 6)))` evaluates to `(+ 3 5 6)`. If `form` does not evaluate to a list, the results are unpredictable, and an error may be signalled.

`..form` This works like `,@` except that the list produced by `form` may be destroyed in the process of creating the result of the backquoted form.

Here's a few examples of backquoted forms showing the results of evaluating them.

```
'(car (1 2 3)) ⇒ (CAR (1 2 3))  
'foobar      ⇒ FOOBAR
```

```
(setq a '(a b c))  
' ,@a      ⇒ Signals an error  
'(1 ,@a 2 3) ⇒ (1 A B C 2 3)  
'(1 ,a 2 3) ⇒ (1 (A B C) 2 3)
```

```
(setq a 'doghead b '(foo to you 2) c 98)  
'(c ,a ,(caddr b) ,(- c 11) ,@(cdr b) a) ⇒ (C DOGHEAD (2) 87 TO YOU 2 A)
```

Using the backquote, we could rewrite our stupid macro as

```
(defmacro stupid (arg1 arg2)  
  '(the-name-of-a-function-thats-much-too-long ,arg1 ,arg2)  
  )  
  
(macroexpand-1 '(stupid (+ 1 a) foo)) ⇒  
  (THE-NAME-OF-A-FUNCTION-THATS-MUCH-TOO-LONG (+ 1 A) FOO)
```

Notice that the arguments are not evaluated at expansion time. Within the body of the macro, the name of a parameter evaluates to the form that was in the corresponding argument position in the macro call.

---

## Examples

One of the best ways to get started writing macros is to examine some that other people have written. This section lists and explains the definitions of several macros.

### Multiple-Value-Setf

Common Lisp defines a function `multiple-value-setq` for setting a list of variables to the values returned by a form. This is only good for setting variables; you cannot use the generalized variable facility as with `setf`. You might try to write a function to serve this purpose, but you would fail miserably, because if you tried to pass the form that returns the values you want to use, it will be evaluated and the values lost. You need to write a macro so that the form will not be evaluated until you want it to be.

---

#### NOTE

This definition of `multiple-value-setf` is not strictly correct in that some tricky order of evaluation concerns are not addressed for the sake of simplicity and exposition.

---

```
;;;
;;; Multiple-Value-Setf takes a list of generalized variable
;;; references and a form, evaluates the form and setf's the
;;; variable references to the corresponding value returned by
;;; the form. If there are more values than places, the extra
;;; values are discarded. If there are extra places, they are
;;; left unchanged.
;;;
;;;
(defmacro multiple-value-setf (placelist form)
  ;; This part is executed at expansion time
  (let (varlist setflist)
    (dolist (place placelist)
      (let ((var (gensym)))
        (setf varlist (append varlist (list var)))
        (setf setflist (append setflist (list place var)))
      )
    )
    ;; This is the form to be substituted for the call
    ;; then evaluated at run time
    '(multiple-value-bind ,varlist ,form
      (setf ,@setflist)
    )
  )
)
```

The expansion of a call to `multiple-value-setf` is a `multiple-value-bind` form that binds a list of variables (`varlist`) to the values returned by `form`, and then does a `setf` that sets the specified places to the values of the variables.

The lists `varlist` and `setflist` are generated in the `dolist` loop. For each place in `placelist`, a variable name is generated with `gensym` and appended to `varlist`. Then the current place and the generated variable are appended to `setflist`, a list of alternating places and variable names. When `multiple-value-setf` is called, the call is replaced with the `multiple-value-bind` form which is then evaluated (at execution time).

It is important to notice the use of `gensym` in the definition of `multiple-value-setf`. Here it is almost essential because you need an indeterminate number of variable names. Another good reason for using `gensym` to generate the names of any variables that will be present in the expansion of a macro call is to avoid collisions with variables that may be present in the environment in which the expansion is executed.

## Substr

This example could be implemented as either a function or a macro. Here it is written as a macro to demonstrate using an `&optional` parameter with a macro. Note that the Common Lisp function `subseq` provides a similar capability.

```
;;;
;;; Substr returns the n character substring of s starting at start.
;;; If there aren't enough characters in s, substr returns nil. If
;;; n is not specified, it defaults to the number of characters in
;;; s after start.
;;;
;;;
(defun substr (s start &optional (n '(abs (- (length ,s) ,start))))
  (let ((temp-len (gensym))
        (temp-i (gensym))
        (temp-str (gensym)))
    )
    '(let ((,temp-len ,n))
      (when (<= (+ ,temp-len ,start) (length ,s))
        (let ((,temp-str (make-string ,temp-len)))
          (dotimes (,temp-i ,temp-len)
            (setf (schar ,temp-str ,temp-i) (schar ,s (+ ,start ,temp-i)))
          )
          ,temp-str)
        )
      )
    )
  )
)
```



In this example, the entire body of the `defmacro` is the form to be returned. It essentially follows the way you could write `substr` as a function. The thing to notice is the specification of the default value for `n`. Since the arguments `s` and `start` may be expressions whose values are not available until execution time, the default value of `n` must be a form that computes the number of characters to include in the substring. You might be tempted to leave out the backquote and use

```
(abs (- (length s) start))
```

as the default value of `n`. In general, though, this will not work because the default is computed at expansion time, and the values of `s` and `start` may not be available then. Note the use of a variable to store the value of `n`. This insures that if `n` is not specified in a call, the form that computes the default value is evaluated only once, rather than every time that the value of `n` is needed. Also notice the use of `gensym` to create a variable name for the `dotimes` counter. This avoids conflicts that might arise if, for instance, we just used `i` as the variable name and we called `substr` with a call like `(substr namestring i)`.

# Object-Oriented Programming

---

## Introduction

You may have heard the term **object-oriented programming** before. Like most buzzwords, it means different things to different people. This chapter will define in general terms what we mean by object-oriented programming, and describe how it is supported in HP's implementation of Common Lisp.

As of this writing, the Common Lisp standard does not define facilities for object-oriented programming. Since this is a valuable capability, Hewlett-Packard has chosen to provide the tools that its engineers and researchers have been using to program with objects. The definition of these tools is being considered along with several other proposals as possible standards for objects in Common Lisp. If and when an objects standard for Common Lisp is defined, HP will implement it. HP expects to continue to support use of our current objects system, and to provide appropriate migration tools in order to converge our definition with a future standard.

The major sections of this chapter are as follows:

Introduction	Describes the general concept of object-oriented programming; defines some necessary terms.
Getting Started	Summarizes the basic functions provided in HP's Common Lisp that support object-oriented programming; concludes with a short example that demonstrates the basics.
Inheritance	Describes how to combine different types of objects to create a new type.
Initialization	Details the process of how an object is initialized when it is created.
Universal Methods	Lists the methods that are defined by default at the time an instance type is defined. Also covers a few extensions made to Common Lisp for working with instances.
Redefining Instance Types	Discusses how to rename and undefine instance types.

---

## NOTE

The term **object** is sometimes used to refer to any unit of Lisp data (as in “`x` is an object of type `list`”). This introduction uses **object** to identify a special kind of data object. We will shortly define some terms that avoid the ambiguity of two different meanings for the word **object**.

---

## What's an Object?

First of all, what is an object? An object is a vehicle for abstracting data and the operations that can be performed on that data. You can think of an object as a “black box” whose only visible characteristics are some buttons on the front panel and a slot for inserting data. The buttons select operations that the object can perform, and the slot provides a means of providing input. The important thing is that the inside of the box is hidden from view. Someone who wishes to utilize its capabilities does not need to know how those capabilities are achieved.

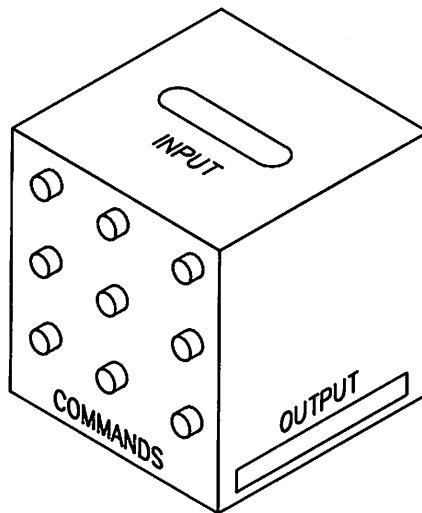



Figure 6-1. An object

You may be saying “But that’s just a subprogram. Every reasonable programming language has those.” The difference is that an object contains data as well as instructions for manipulating it. For instance, in a graphics program, an object could represent a particular figure to be shown on the screen. This object would be made up of data (such as the origin and dimensions of the figure) and the operations that can be performed on that data (such as rotating or scaling). To rotate the figure you need only to press the rotate button and insert the desired angle of rotation.




The beauty of this is that while different kinds of figures in your graphics program may require different algorithms to rotate them, you can press the rotate button on any figure and have the proper operation performed. You don't have to say "Well, this figure is a dodecahedron so I'll use the dodecahedron rotating algorithm." Because the operations are part of the object, it "knows" which algorithm to use to rotate itself. As long as all the objects that represent figures have a rotate button, you're all set.

So we now know that

`data + operations = object`

## Messages

Asking an object to perform an operation (pushing a button on the black box) is referred to as sending the object a message. A message contains the name of the operation and any **external** data the object needs to perform the operation (such as the angle of rotation). Note that the message contains only the **name** of the operation, and not any details as to how the operation is to be done. Thus, different objects can have very different reactions to the same message, though in a useful system the final result should be similar in order to maintain mnemonic consistency.



Think about when a manager asks an employee to perform a task. They usually will tell them:

- What general job is to be done. "Write a manual." (This is the name of the operation.)
- Any necessary auxiliary information. "About Lisp Programming."

Assuming there is more than one employee competent to perform the task, it doesn't matter which one the manager asks to do the job. The manual will be written. However, it is unlikely that any two writers would get the job done in the same way. It is the built-in knowledge of the employees that allows the manager to request the performance of tasks without having to go into great detail. They also do not have to tailor the procedure to each individual.

## Terms

Like most specialized subjects, object-oriented programming has its own set of terms. You've already seen the word "message." Before going any further, let's define a few more terms.

Instance	An object. We will use instance in order to avoid confusion with the other definition of the word object.
Instance Type	A type that describes a kind of instance. Instances of a particular instance type store the same types of data and have the same set of operations defined on them. Sometimes called a <b>flavor</b> .
Method	An operation that can be performed on an instance of a given instance type. These are the "buttons" of an instance.
Instance Variable	An instance type definition contains a template for the data that will be stored in an instance of that instance type. This is similar to a Pascal record, or a structure created with <code>defstruct</code> , in that the template specifies names of fields that store data. In instances, the fields are called <b>instance variables</b> .



---

## Getting Started

There are certain primitive tools needed for object-oriented programming. You must be able to

- Define an instance type.
- Define a method.
- Create an instance.
- Send a message to an instance.

Since these operations are so closely related, it is difficult to describe any of them without mentioning another. Please be patient while reading the following descriptions. All will be revealed by the example that follows them.

The functions that implement objects are defined in the `objects` module, and are named by symbols in the `extn` package. You should put the following forms at the beginning of any programs you write that use objects.

```
(require "objects")  
(use-package 'extn)
```

### Defining an Instance Type

Instance variables are like slots in a structure, except that access to them is allowed only within the methods defined on that type of instance. Anywhere else, they must be accessed indirectly by sending a message to the instance that contains them. This insures that programs that use instances do not rely on the instances' internal data.

An instance type consists of a description of its instance variables, and the definitions of its methods. The first step in defining an instance type is describing the data (instance variables) to be stored in instances of that type. You do this with the macro `define-type`. Do not be intimidated by the large variety of options available for `define-type`. You do not need to immediately understand them all to start programming with instances.

```
(extn:define-type type-name [doc-string] Macro  
  {(:var var {var-option}*) | define-type-option}*)
```

The name of the created instance type is *type-name*; it should not conflict with any of the system defined types (such as `integer` or `list`). A call to `define-type` returns the symbol that is the name of the instance type that was defined. If specified, *doc-string* is installed as the type documentation for the symbol *type-name*.

## Specifying Instance Variables

The instance variables of the type are specified with forms like

```
(:var var {var-option}*)
```




The name of the instance variable is *var*. Zero or more *var-options* may be specified. They control various properties of the instance variable. Valid *var-options* are:

- |                           |  |
|---------------------------|--|
| <code>(:init form)</code> | When a new instance of the type being defined is created, <i>form</i> is evaluated and the value it returns is the initial value of the affected instance variable. See the “Initialization” section of this chapter for more detail.  |
| <code>(:type type)</code> | This declares the affected instance variable to be of type <i>type</i> .   |
| <code>:gettable</code>    | This option sets up a parameterless method named <i>:var</i> that returns the value of the instance variable <i>var</i> . This is how programs can access the values of instance variables. Note that this does not hinder the ability of an instance to hide its internal data since the values are accessed by sending a message. If the internal representation were to change, the methods created by the <code>:gettable</code> option could be rewritten to simulate the old versions. (An example of this appears later in this chapter.) |
| <code>:initable</code>    | If an instance variable is defined with the <code>:initable</code> option, then you may specify an initial value for it when creating an instance of that instance type.   |
| <code>:settable</code>    | This sets up a method named <code>:set-var</code> that takes one parameter and sets the value of the affected instance variable to that parameter. A <code>:settable</code> instance variable is always made <code>:initable</code> and <code>:gettable</code> . If this is not desired, you must define your own method to set the instance variable, instead of using the <code>:settable</code> option, or use the <code>:redefined-methods</code> option to suppress creation of the undesired methods.                                      |

## Define-Type Options

You can also specify options that globally affect the instance type being defined. These are valid in the same places within a call to `define-type` as instance variable definitions. Stylistically, it is preferable to segregate `define-type` options and instance variable definitions. The only option that may appear more than once in a single call to `define-type` is `:inherit-from`.

The available options are:

- 
- :all-gettable** All instance variables of this instance type are **:gettable**. This does not apply to inherited instance variables.
- :all-settable** All instance variables of this instance type are **:settable**. This does not apply to inherited instance variables.
- :all-initable** All instance variables of this instance type are **:initable**. This does not apply to inherited instance variables.
- (:fast-methods  
{method-name}\*)** In the current implementation, this option has no effect. The methods for this instance type named by the symbol(s) *method-name* (which might not be defined at this time) are made more efficient to invoke. This could make other methods less efficient, so should only be used when there are a few methods that are used substantially more often than the others. If no symbols are given, then this option has no effect.
- 
- (:inline-methods  
{method-name}\*)** In the current implementation, this option has no effect. This option specifies that the methods for this instance type named by the symbol(s) *method-name* (which might not be defined at this time) should be processed inline so that the overhead of a call is avoided. The compiler may ignore this option. This option trades increased code size for an increase in execution speed. If no symbols are given, then this option has no effect.
- (:notinline-methods  
{method-name}\*)** This option specifies that the methods for this instance type named by the symbol(s) *method-name* (which might not be defined at this time) should **not** be processed inline. The compiler must obey this directive, but is free to process inline any methods not listed within this option. If no symbols are given, then this option has no effect.
- 
- (:redefined-methods  
{method-name}\*)** **define-type** automatically defines some methods for the instance type being defined. These include a number of “universal” methods, methods resulting from **:settable** and **:gettable** options, and inherited methods. Any method named in the **:redefined-methods** option will not be created by **define-type**. This option should be used when there are automatically created methods (universal or inherited) that you plan to redefine. Specifying these in a **:redefined-methods** option prevents them from being accidentally redefined if the **define-type** is reexecuted.



`(:init-keywords  
{keyword}*)`

This declares that the given *keywords* are valid initialization keywords for calls to `make-instance` (the function for creating an instance) that create instances of the type being defined. See the “Initialization” section later in this chapter.

`:no-init-keyword-check`

When this option is given to `define-type`, `make-instance` will not check the validity of initialization keywords when creating an instance of the type being defined.

`(:inherit-from  
instance-type-name  
{inherit-option}*)`

The `:inherit-from` option is used to add characteristics of existing instance types to the instance type being created. This is a complicated topic, and is covered in the “Inheritance” section of this chapter.

### Example

The following code sets up an instance type whose members represent Cartesian vectors.

```
(extn:define-type vector-instance
  ;; Instance Variables
  (:var theta (:type float))
  (:var magnitude (:type float))
  ;; Options
  :all-settable
)
```

The two instance variables are declared to be of type `float`. The `:all-settable` option causes `define-type` to create the methods

- `:theta` and `:magnitude` for accessing the values of the two instance variables.
- `:set-theta` and `:set-magnitude` for updating the values of the instance variables.

## Defining a Method

To define a method for an instance type, use the `extn:define-method` macro. This is similar to `defun`. The parameters of `define-method` include the name of the method, the name of the instance type the method applies to, a description of the parameters, and the actions the method should take.

```
(extn:define-method (instance-type-name method-name) lambda-list           Macro
  {declaration}*
  {form}*)
```

The instance type *instance-type-name* must exist at the time `define-method` is processed. Otherwise, an error results. A call to `define-method` returns a list of two symbols: the instance type name symbol and the method name symbol.

The name of the method can be any symbol, but it is suggested that you use keywords (i.e., symbols in the keyword package).

The *forms* (as in a `defun`) are evaluated when the method is invoked. The method returns the value of the last form evaluated during its invocation. The forms within the method definition can access the instance variables of *instance-type-name* just by using their names as if they were lexical variables. The forms may also refer to the lexical variable `self`, which refers to the instance upon which the method was invoked.

Here's an example call to `define-method` that uses the `vector-instance` instance type defined above:

```
;;;
;;; Method that returns x-coordinate of a Cartesian vector instance
;;;
(extn:define-method (vector-instance :x-coordinate) ()
  (* magnitude (cos theta))
  )
```

Note that we are now able to partially access a `vector-instance` as if it was represented as an x-y pair (rectangular) instead of a magnitude-theta pair (polar). This will be expanded upon in a forthcoming example.

## Creating an Instance

The most direct means of creating an instance is the function `extn:make-instance`.

```
(extn:make-instance instance-type-name {init-keyword value}*) Function
```

A call to `make-instance` returns a new instance belonging to the named instance type (which must exist and be an instance type). When the instance is created, a universal method named `:initialize` is invoked on it with all the *init-keyword value* pairs as parameters. Typically, the *init-keywords* are the names of initable instance variables preceded by a `:`, and the values are the initial values for those instance variables. If the keywords are not valid initialization keywords and the instance type was not defined with the `:no-init-keyword-check` option, then an error occurs. For information on the exact initialization process, see the “Initialization” section of this chapter.

Here’s an example of a call to `make-instance`:

```
(setq my-vector (extn:make-instance 'vector-instance
                                   :theta (/ pi 2)
                                   :magnitude 67))
```

## Sending Messages

An operation may be invoked on an instance by sending it a message. There are two functions that do this:

The commonly used one is `extn:=>` (pronounced “send”).

```
(extn:=> instance method-name {argument}*) Function
```

If *method-name* is not a keyword, it must be quoted. Arguments to the method are simply added after *method-name*. The form

```
(extn:=> my-vector :set-theta pi)
```

would return the results of invoking method `:set-theta` on the instance `my-vector` with the value `pi`. This particular method has the side effect of changing the value of `theta` for that instance to the value of its argument (in this case, `pi`). If the instance type of the instance being sent to does not support the method, an error is signalled.

A similar function is `extn:send?`.

`(extn:send? instance method-name {argument}*)`

*Function*

This works identically to `=>` except that if there is no method *method-name* for the instance type of *instance*, or *instance* is not an instance, no error is signalled and `nil` is returned.

You can explicitly test whether or not an instance supports a particular method with the function `extn:supports-operation-p`.

`(extn:supports-operation-p object method-name)`

*Function*

This returns true only if *object* is an instance, and it has a method *method-name* defined on it.

## A Brief Tutorial

This simple example shows the concepts discussed above in action. It details creating an instance type named `vector-instance`, methods for operating on `vector-instance` instances, and functions that use those methods. In the example you will

1. Define the instance type.
2. Define the primitive operations (methods) for the instance type.
3. Define some functions to use the instances.
4. Create some instances and test the methods and functions.

### Defining the Instance Type

The first thing to do when programming with objects is to define the instance types you'll be using. In this example we will be dealing with the `vector-instance` instance type. The Cartesian vectors will be stored in polar (angle-magnitude pairs) form so our instance type will have two instance variables: `theta` and `magnitude`. We'll assume that the default angle (in radians) is 0, so that is the default initialization for `theta`. We want to be able to specify initial values when creating a Cartesian vector, as well as have default methods set up to access and update both of the instance-variables, so the `all-settable` option is specified.

```
(require "objects") ; The rest of this example assumes that
(use-package 'extn) ; these two forms have been evaluated.
```

```
(define-type vector-instance
  ;; Instance Variables
  (:var theta (:type float)
             (:init 0))
  (:var magnitude (:type float))
  ;; Options
  :all-settable
)
```

## Choosing and Defining Methods

Once you have defined the shape of the data in your object (by executing the `define-type` form), you must complete the definition of the object by creating the operations that are part of it.

What kind of operations do we want to define for the `vector-instance` instance type? Since the instance-variables are settable, we already have methods `:magnitude`, `:set-magnitude`, `:theta`, and `:set-theta` (as well as some initialization methods). We probably want to be able to multiply the Cartesian vector by a number. Let's call this method `:scale`.

```
;;;
;;; The scale method multiplies a vector-instance by some number x
;;;
(define-method (vector-instance :scale) (x)
  (setq magnitude (* x magnitude)))
```

Notice that the `magnitude` instance-variable is referenced just by using its name, and that it can be `setq`d.

## Another Way of Looking at Vectors

There are two common ways of specifying Cartesian vectors. One is the way we're representing them in our `vector-instance` instance type. These angle-magnitude pairs are referred to as polar coordinates. The other way to represent a vector is to store the values of its `x` and `y` coordinates. These are known as rectangular coordinates.

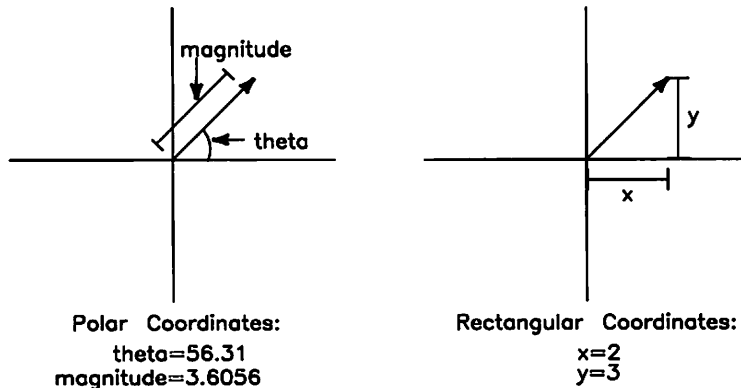


Figure 6-2. Two Ways to Represent a Vector

Suppose there are situations where it's more convenient to look at a vector-instance as rectangular coordinates than as polar coordinates. We could define another instance type to handle this situation, but then we'd be defeating the whole purpose of programming with instances. Why not just define methods that "pretend" that the vector is stored as x and y coordinates? We'll need methods :x and :y (to obtain the values of the coordinates), and :set-x and :set-y (to update the vector based on x and y coordinates).

```
;;;
;;; Return the x coordinate of a vector
;;;
(define-method (vector-instance :x) ()
  (* magnitude (cos theta)))

;;;
;;; Return the y coordinate of a vector
;;;
(define-method (vector-instance :y) ()
  (* magnitude (sin theta)))

;;;
;;; Set the x coordinate of a vector
;;;
(define-method (vector-instance :set-x) (new-x)
  (setq magnitude (sqrt (+ (expt new-x 2)
                           (expt (=> self :y) 2))))
  (setq theta (atan (=> self :y) new-x)))

;;;
;;; Set the y-coordinate of a vector
;;;
(define-method (vector-instance :set-y) (new-y)
  (setq magnitude (sqrt (+ (expt (=> self :x) 2)
                           (expt new-y 2))))
  (setq theta (atan new-y (=> self :x))))
```

Once these methods are defined, anyone using a `vector-instance` object can access it with polar or rectangular coordinates. There is no need for external conversion routines, since the `vector-instance` object "knows" both forms of the Cartesian vector. Instead of building in additional data, we have built in the means to extrapolate this data.

However, one place where we know that the vector is not actually stored as rectangular coordinates is in the methods that are part of the instance. The main symptom of this is that within a `vector-instance` method we cannot refer to the x and y coordinates by just naming them. We must invoke the method :x or :y on `self`.

Now let's define a function that adds two vectors and returns a `vector-instance` object representing their sum.

```
;;;
;;; Return the sum of two Cartesian vectors
;;;
(defun vector-instance-add (vector1 vector2)
  (let ((new-vector (make-instance 'vector-instance)))
    (=> new-vector :set-x (+ (=> vector1 :x) (=> vector2 :x)))
    (=> new-vector :set-y (+ (=> vector1 :y) (=> vector2 :y)))
    new-vector))
```

Being able to reference the vectors as x-y pairs makes this function much simpler than if we would've had to add the vectors in polar form. This function points out one flaw in our pretext that the vector is stored as rectangular coordinates: we can't use initialization keywords for x and y in our call to `make-instance`. There are ways of doing this however; see the "Initialization" section for details.

## Another Short Example

Here is a simple example that uses the object-oriented programming features of HP's Common Lisp to implement a computerized bank account. This first section of code defines the `bank-account` instance type, and a few basic methods and functions for manipulating bank accounts.

```
;;;
;;; An instance type to represent bank accounts
;;;
(define-type bank-account
  (:var holder (:type simple-string))
  (:var acct-num)
  (:var balance (:type number))
  :all-initable
  :all-gettable)

;;;
;;; Function for creating new accounts
;;;
(defun open-account (name number initial-balance)
  (if (and (simple-string-p name)
          (numberp initial-balance)
          (> initial-balance 0))
      (make-instance 'bank-account
                     :holder name
                     :acct-num number
                     :balance initial-balance)
      (error "Bad name: ~A or Balance: ~A " name initial-balance))
  )

(setq acct1 (open-account "Bobby Brown" '555-55-5555 100.00))
(=> acct1 :balance) => 100.0

;;;
;;; Method to make a deposit to an account. The new balance
;;; is returned.
;;;
(define-method (bank-account :deposit) (amount)
  (if (and (numberp amount) (> amount 0))
      (setf balance (+ balance amount))
      (error "Bad deposit amount ~A" amount))
  )

(=> acct1 :deposit 50) => 150.0
(=> acct1 :balance) => 150.0
```



```
;;;
;;; Method to make a withdrawal from an account. The new
;;; balance is returned.
;;;
(define-method (bank-account :withdraw) (amount)
  (cond ((or (not (numberp amount)) (< amount 0))
        (error "Improper Withdrawal Amount ~A" amount))
        ((< balance amount)
         (error "Insufficient Funds -- Transaction denied"))
        (t (setf balance (- balance amount))))
  )
)

(=> acct1 :withdraw 25) => 125.0
(=> acct1 :balance) => 125.0
```

---

## Inheritance

The object-oriented programming features of HP's Lisp allow you to create a new instance type by combining previously defined instance types with any additional instance variables and methods you care to define. This mechanism is called **inheritance** since the new type inherits the characteristics of the previously defined type(s).

As an example of where inheritance might be used, consider a graphics program that deals with geometric shapes. The programmer might define a **shape** instance type that implements the instance variables and methods needed for all shapes, and then define more specialized instance types that inherit from **shape**, such as **circle** or **rectangle**. This way, the code that is common to all shapes only needs to be written once. We could even go one step further and define an instance type **colored-rectangle** which inherits from **rectangle** which inherits from **shape**.

For convenience, we define the following: If instance type **a** is inherited by instance type **b**, then **a** is a **parent** type with respect to **b**, and **b** is a **child** type with respect to **a**.

### Defining a Type that Inherits

The `:inherit-from` option to `define-type` indicates which (if any) existing instance types should be inherited by the type being defined. The following code defines two instance types: **circle**, and **colored-circle** (which inherits from **circle**).

```
(extn:define-type circle
  (:var origin)
  (:var radius (:type float))
  (:var exposed?)
  :all-settable)

(extn:define-type colored-circle
  (:var color :settable)
  (:inherit-from circle :init-keywords)
)
```

The `:inherit-from` option has the following form:

```
(:inherit-from inherited-type-name {inherit-option}*)
```

The valid values for *inherit-option* are:

<code>(:methods [:except {method-name}])</code>	Default behavior is for an instance type to inherit all of the methods (except universal methods) defined on the type(s) it inherits from. If there is a <code>:methods</code> option to <code>:inherit-from</code> and <code>:except</code> is omitted, only the named methods are inherited. If <code>:except</code> is specified, then the instance type inherits all of the inherited type's non-universal methods except those named.
<code>:init-keywords</code>	Default behavior is to <b>not</b> allow the use of initialization keywords that are defined for an inherited type. This option declares that all legal initialization keywords defined for the inherited type are also legal for the inheriting type. See the "Initialization" section for details.
<code>(:init-keywords :except {keyword}*)</code>	Default behavior is to <b>not</b> allow the use of initialization keywords that are defined for an inherited type. This option declares that all legal initialization keywords defined for the inherited type, except those named in the option, are also legal for the inheriting type. See the "Initialization" section for details.
<code>(:variables {ivar-name   (ivar-name alias)}*)</code>	This option allows methods to access and update the named inherited instance variables as if they were defined directly in the inheriting type. (Normally they must be accessed by invoking a method). The <i>(ivar-name alias)</i> variation of the option allows methods of the child instance type to access the inherited instance variable <i>ivar-name</i> with the name <i>alias</i> . See "Pseudo Instance Variables" below for more information.

## Inheriting Methods

By default, a child type inherits all of its parents' (there may be more than one) methods, except for their universal methods. If this is not what you want, the `:methods` option to the `:inherit-from` clause can be used to specify exactly which methods are inherited. For efficiency, you should only inherit methods that will be used by the inheriting type. Naming a method in the `:redefined-methods` option to `define-type` prevents it from being inherited, but the preferred way to do this is the `:methods` option. If you attempt to inherit more than one method with the same name, or try to inherit a method with the same name as a method resulting from a `:settable` or `:gettable` option, an error is signalled.

An inheriting type only inherits the methods defined on its parents at the time of its definition. Defining a new method on a parent type after the `define-type` for the child has been evaluated has no effect on the child type. The new method does not automatically propagate to the child; to inherit the method, the `define-type` for the child must be reevaluated.

## Accessing Inherited Methods

Inherited methods are invoked just like any other method. Here's an example:

```
(require "objects")
(use-package 'extn)
;;;
;;; Animal Instance Type
;;;
(define-type animal
  (:var habitat :settable)
  (:var weight :settable)
  (:var year-of-birth :initable :gettable)
  (:var living? (:init t) :settable)
  (:var year-of-death :settable))

;;;
;;; Method to return animal's age in solar years
;;; Note: Assumes years are A.D.
;;;
(define-method (animal :age) ()
  (if living?
    (- (current-year) year-of-birth)
    ;; else
    (- year-of-death year-of-birth)
  )
)

;;;
;;; Race-horse instance type (inherits from animal)
;;;
(define-type race-horse
  (:inherit-from animal :init-keywords
    (:variables living?))
  (:var offspring :settable)
  (:var races :settable))

(setq man-of-peace (make-instance 'race-horse :year-of-birth 1967))

(current-year) => 1985 ; This function is not predefined
(=> man-of-peace :age) => 18
```

The inherited methods `:set-year-of-birth` and `:age` can be accessed by an instance of type `race-horse` as if they were defined specifically for that type. If we want to define a more specific age method for `race-horse` (perhaps one that gives the horse's equivalent human age) we can.

```
;;;
;;; Method to return race-horse's equivalent human age
;;;
(define-method (race-horse :age) ()
  (let ((birth-year (=> self :year-of-birth))
        real-years)
    (if living?
        (setf real-years (- (current-year) birth-year))
        ;; else
        (setf real-years (- year-of-death birth-year)))
    (* real-years 3) ;; Horses live ~1/3 as long as humans
  )
)

(=> man-of-peace :age) => 54
```

After this method is defined, invoking the `:age` method on an instance of type `race-horse` will execute the new code. After adding this new method, we should add a `:methods` option to the `:inherit-from` clause in the `define-type` for `race-horse` so that we do not inherit the `animal :age` method. Then if we ever need to reevaluate the `define-type`, we won't also have to reevaluate the definition of the specialized age method for `race-horses`.

### Resolving Method Ambiguity

There are some problems associated with redefining inherited methods. For instance, what if we want to define an `animal` method that returns T if the animal has lived longer than a given number of years, and we want the years to be normal solar years? The natural way to define such a method would be

```
(define-method (animal :lived-longer-than-p) (number-of-years)
  (> (=> self :age) number-of-years)
)
```

If we use this method with a `race-horse` instance (or any other inheritor of `animal` that defines its own `:age` method), we will not get the desired result. We need a way to specify that we want to invoke the `:age` method defined for `animal` instances.

`(call-method method-name {arg}*)`  
`(apply-method method-name {arg}* list)`

Macro  
Macro

The macros `call-method` and `apply-method` are used within the body of a `define-method` for some type *t* to invoke a method defined on type *t*. *Method-name* is not evaluated, it must be the name of a method defined on type *t* (either directly or inherited). With `call-method`, the remaining forms are evaluated and passed as arguments to the method being invoked. With `apply-method`, the *args* are evaluated to produce individual arguments and *list* is evaluated to produce a list of arguments; the invoked method is passed the individual arguments followed by the elements of the list. (`call-method` is analogous to `funcall` and `apply-method` is analogous to `apply`.) Within the invoked method, `self` refers to the same instance that it did in the method containing the `call-method` (or `apply-method`).

With `call-method` we can write the `:lived-longer-than-p` method as follows and have it work correctly.

```
(define-method (animal :lived-longer-than-p) (number-of-years)
  (> (call-method :age) number-of-years)
)
```

### Accessing Methods that are Not Inherited

It is sometimes desired within the definition of a method for type *x* to invoke a method defined in a parent type of *x* without actually inheriting the method. For instance, it may be possible to reuse code by defining the child's method in terms of the parent's.

To do this, there is another variety of call to `call-method` and `apply-method`.

`(call-method (instance-type-name method-name) {arg}*)`  
`(apply-method (instance-type-name method-name) {arg}* list)`

Macro  
Macro

This form is used within the body of a `define-method` to invoke a method that is defined on a **directly** inherited type. If the `call-method` appears in a `define-method` for type *t*, then *instance-type-name* must be the name of an instance type that appears in an `:inherits-from` option to the `define-type` for *t*. Additionally, *method-name* must be defined for *instance-type-name* when the `call-method` is executed. Within the invoked method, `self` refers to the same instance that it did in the method containing the `call-method` (or `apply-method`).

Here's a demonstration of using call-method to implement a child's method by augmenting an existing method that is defined on a parent.

```
(require "objects")
(use-package 'extn)
;;;
;;; The basic employee instance type
;;;
(define-type employee
  (:var name :initable :gettable)
  (:var address :settable)
  (:var phone :settable))

;;;
;;; Method to display basic employee info
;;;
(define-method (employee :display-info) ()
  (format t "Name: ~A~%Address: ~A~%Phone: ~A~%" name address phone)
  )

;;;
;;; A restricted access employee type
;;;
(define-type secured-employee
  (:inherit-from employee :init-keywords
    (:methods :except :display-info)
    (:variables name address phone))
  (:var password))

;;;
;;; Method to display basic info if password allows access
;;; (Assumes that the function request-and-get-password exists)
;;;
(define-method (secured-employee :display-info) ()
  (if (equal (request-and-get-password) password)
      (call-method (employee :display-info))
      (error "Access Denied")
  )
  )
)
```

## Inheriting Instance Variables

A child instance type inherits all of its parents' (there may be more than one) instance variables. However, these instance variables are not directly accessible in the child's methods; they must be accessed by invoking the parents' access methods. For instance, in a method for the `colored-circle` instance type, we cannot access the inherited instance variable `exposed?` as if it were a lexical variable.

```
(define-type circle
  (:var origin)
  (:var radius (:type float))
  (:var exposed?)
  :all-settable)

(define-type colored-circle
  (:var color :settable)
  (:inherit-from circle :init-keywords)
  )

(define-method (colored-circle :show-yourself)
  (if (not (=> self :exposed?))      ; :exposed must be accessed
      (prog (=> self :set-exposed? T) ; with a method.
            (=> self :dither-expose))
      ;; Else
      (=> self :refresh)))
```

Since `colored-circle` inherits the methods `:exposed?` and `:set-exposed?` from `circle`, we can use them to access and update the value of the inherited instance variable `exposed?`.



## Name Conflicts

Inheriting more than one instance variable with the same name is allowed (there will be two distinct variables), but you can only have one method with a particular name. This means that if the parents' have methods with the same names, you will get an error unless you make sure to inherit only one of the identically named methods.

```
(define-type a
  (:var x)
  :all-settable)

(define-type b
  (:var x)
  :all-initable
  :all-gettable)

;;;
;;; This is the WRONG way to inherit from both a and b.
;;; We are trying to inherit two methods named :x.
;;;
;(define-type c
;  (:inherit-from a)
;  (:inherit-from b)
;  (:var z)
;  :all-gettable)

;;;
;;; This is the right way to do it. We must specifically ask
;;; not to inherit one of the :x methods.
;;;
(define-type c
  (:inherit-from a)
  (:inherit-from b (:methods :except :x))
  (:var z)
  :all-gettable)
```

To access the second instance variable named `x`, we must define our own method to do so using `call-method`.

```
;;;
;;; Method to get to type b's x instance variable
;;;
(define-method (c :x-b) ()
  (call-method (b :x))
)
```

The same conflict shows up in the situation where type *b* inherits from type *a*, and type *c* inherits from both type *a* and type *b*. In this case, *c* will have two copies of *a*'s instance variables, but you must explicitly not inherit some methods to avoid name conflicts. This is not a situation that you should attempt to emulate. It is described here for completeness.

```
(define-type a
  (:var x)
  :all-settable)

(define-type b
  (:var y)
  (:inherit-from a))

(define-type c
  (:var z)
  (:inherit-from a)
  (:inherit-from b (:methods :except :x :set-x)))

(define-method (c :set-x-b) (value)
  (call-method (b :set-x) value ))

(define-method (c :x-b) ()
  (call-method (b :x)))

(setq zone (make-instance 'c))
(=> zone :set-x 5) => 5
(=> zone :x) => 5
(=> zone :set-x-b 9) => 9
(=> zone :x) => 5
(=> zone :x-b) => 9
```

### Pseudo Instance Variables

You can simulate direct access to inherited instance variables with the `:variables` option to `:inherit-from`. This option provides a shorthand notation for invoking certain methods that are defined for the parent instance type (the methods may or may not be inherited by the child type).

The `:variables` option has the following form:

```
(:variables {ivar-name | (ivar-name alias)}*)
```

When this option is specified, methods for the child type may use *ivar-name* (or *alias* if the second form of the option is used) as a shorthand notation for the form

```
(call-method (parent-type :ivar-name))
```

In addition, methods for the child type can use

```
(setq ivar-name value) or  
(setq alias value) (if the second form of the :variables option is used)
```

as a shorthand for

```
(call-method (parent-type :set-ivar-name) value)
```

The parent instance type must have the appropriate method(s) defined on it (`:ivar-name` and `:set-ivar-name`) in order to use the shorthand.

### Example of the `:variables` Option

```
(define-type human  
  (:var birthdate :initable :gettable)  
  (:var height :settable)  
  (:var weight :settable)  
  (:var name :initable :gettable))  
  
(define-type doctor  
  (:var alma-mater :initable :gettable)  
  (:var specialty :settable)  
  (:inherit-from human  
    (:variables height  
                 (weight tonnage) ; tonnage is an  
                               ) ; alias for weight  
    :init-keywords)  
  )
```

With these instance type definitions, we can define methods for `doctor` that can refer to the inherited instance variable `height` and the alias `tonnage` as if they were instance variables defined directly in the `doctor` instance type.

```
(define-method (doctor :change-size) (w-delta h-delta)  
  (setq tonnage (+ tonnage w-delta))  
  (setq height (+ height h-delta))  
  )
```

Note that the `:variables` option does not define any methods. Using this option does not by itself allow instance types that inherit from `doctor` to access the same variables. To make access possible by types that inherit from `doctor`, the necessary methods must be inherited by `doctor`. In this particular case they are, since there are no `:methods` options in the `:inherit-from` clause; all `human` methods are inherited by `doctor`. Specifying an alias in a `:variables` option does not change the name of the inherited method that accesses the aliased instance variable (eg. in the example above, `doctor` inherits a method named `:weight`, not one named `:tonnage`).

---

## Caveats

The behavior of the macros that implement object-oriented programming can surprise you. For instance, a call to the following function will not correctly define the instance type `fromfoo`.

```
(defun save-time ()
  (extn:define-type foo
    (:var a)
    (:var b (:type simple-string))
    :all-settable)
  (extn:define-type fromfoo
    (:var gonzo :settable)
    (:inherit-from foo)
  )
)
```

If you try to invoke one of the methods that are supposed to be inherited from `foo` (such as `:a`) on an instance of type `fromfoo`, you will get an error message saying that there is no such method for `fromfoo`. This happens because of the way the `define-type` macro is implemented. It does things both at macro expansion time and at run time. When the macro call to define `fromfoo` is expanded, not all of the information it needs to work correctly is available.

The moral of the story is: The macros that implement object-oriented programming may not behave as expected if you call them anywhere besides at top level. If you have an unexplained bug in code that deals with instances, this is a good place to look for it.

For more information on errors of this kind, see the “Preprocessing” section of the “Concepts” chapter.

---

## Initialization

When `make-instance` is called to create a new instance, it begins a sequence of actions to initialize the instance variables of the new object.

1. Check that the initialization keywords in the keyword-value list passed to `make-instance` are legal. If there are any that are not recognized, an error occurs.
2. Invoke the universal method `:initialize` on the uninitialized instance, passing it the keyword-value list.
  - a. The `:initialize` method first invokes the `:initialize` methods for all instance types directly inherited by the type of the instance being initialized. The keyword-value list is also passed to these methods.
  - b. The `:initialize` method then invokes the universal method `:initialize-variables` on the new instance, passing it the keyword-value list. For any keyword that matches the name of an `:initable` instance variable, the corresponding value is assigned to that instance variable.

Then, any instance variable that remains uninitialized is assigned the default value (if any) that was specified in the `define-type` for this instance type. This is done in the order in which the instance variables appear in the `define-type`. The forms that specify the default values are evaluated in the context of the `:initialize-variables` method, so they may refer to other instance variables as lexical variables.

- c. Finally, from within `:initialize`, the method `:init` is invoked, passing it the keyword-value list. The default definition of `:init` is empty. It is provided so that you can define arbitrary initialization code for a particular instance type.

Here's some code that shows how the initialization methods for a particular instance type **might** be defined.

```
;;;
;;; The instance type definition
;;;
(define-type child
  (:var x1 :initable)
  (:var x2 (:init 0) :initable)
  (:var x3 (:init (foo x1 x2)))
  (:inherit-from parent1)
  (:inherit-from parent2)
)

;;;
;;; Example definition of :initialize method
;;;
(define-method (child :initialize) (keylist)
  (call-method (parent1 :initialize) keylist)
  (call-method (parent2 :initialize) keylist)
  (call-method :initialize-variables keylist)
  (call-method :init keylist)
)

;;;
;;; Example definition of :initialize-variables method
;;;
(define-method (child :initialize-variables) (keylist)
  ;; Initialize :initables according to keylist
  (do* ((unprocessed-keys keylist (cddr unprocessed-keys))
        (keyword (car unprocessed-keys) (car unprocessed-keys))
        (value (cadr unprocessed-keys) (cadr unprocessed-keys)))
    ((null unprocessed-keys)
     (case keyword
       ((:x1) (setq x1 value))
       ((:x2) (setq x2 value)))
    )
  ;; Assign defaults if currently unassigned
  (unless (assignedp x2) (setq x2 0))
  (unless (assignedp x3) (setq x3 (foo x1 x2)))
)

;;;
;;; Empty definition of :init
;;;
(define-method (child :init) (keylist)
)
```

Note the use of `assignedp`. This is an HP extension to Common Lisp for dealing with instances.

`(assignedp instance-variable)`

*Function*

A call to `assignedp` returns true if *instance-variable* has been previously assigned a value. An error is signalled if `assignedp` is called with an argument that is not an instance variable. Consequently, `assignedp` can only be called from within a method.

## Custom Initializations

In a previous example that dealt with a Cartesian vector instance type, we wanted to specify initial values for x and y coordinates in a call to `make-instance`, but couldn't because there were no corresponding instance variables. The following example shows how to do this by writing an `:init` method for `vector-instance`.

```
;;;
;;; The instance type definition
;;;
(define-type vector-instance
  ;; Instance Variables
  (:var theta (:type number)
    (:init 0))
  (:var magnitude (:type number))
  ;; Options
  :all-settable
  (:init-keywords :x :y) ; This is new
)

;;;
;;; :init method for vector-instance to allow initializing x and y
;;;
(define-method (vector-instance :init) (keylist)
  (let ((xvalue (getf keylist :x))
        (yvalue (getf keylist :y)))
    ;; Both x and y must have values specified
    (cond ((and xvalue yvalue)
           (setq magnitude (sqrt (+ (expt xvalue 2)
                                    (expt yvalue 2))))
           (setq theta (atan yvalue xvalue)))
          )))
)
```

Now you can create a new `vector-instance` instance and initialize it as if it had `x` and `y` instance variables.

```
(setq vector1 (make-instance 'vector-instance :x 4.0 :y 3.0))
```

Note that our `:init` method is not robust. It assumes that you never want to initialize `x` or `y` to `nil` (a reasonable assumption in this case, but not in others). Also, there are no checks to see if initial values were given for `magnitude` and `theta`, as well as `x` and `y`. A real implementation would need to specify which values take precedence, or signal an error when illogical combinations of initialization keywords are given.



---

## Universal Methods

Whenever a `define-type` is executed, several methods are automatically defined for the new instance type. These are known as **universal methods** (since they are universal to every instance type). If you do not want a particular universal method to be automatically defined for an instance type, then that method should be named in a `:redefined-methods` option to `define-type`.

To avoid conflicts when a type inherits from more than one other type, universal methods are not inherited unless explicitly specified in the `:inherits-from` option. In many cases, inheriting universal methods will not produce the desired results.

The universal methods are as follows:

<code>:print</code> <i>output-stream depth</i>	The <code>:print</code> method is invoked by various printing functions when an instance is to be printed. It is not normally invoked directly from user code. <i>Output-stream</i> is the stream to send the output to, and <i>depth</i> is the current level of print nesting.
<code>:describe</code>	The <code>:describe</code> method displays an instance. It is not normally invoked from user code. The default definition prints a description of <code>self</code> to <code>*standard-output*</code> . The instance variables printed are those of the instance type of <code>self</code> and any types that it inherits.
<code>:initialize</code> <i>keylist</i>	This is the basic initialization method of an instance type. <i>Keylist</i> is an alternating list of initialization keywords and values. See the “Initialization” section for more detail.
<code>:initialize-variables</code> <i>keylist</i>	Initializes the instance variables of an instance. <i>Keylist</i> is an alternating list of initialization keywords and values. See the “Initialization” section for more detail.
<code>:init</code> <i>keylist</i>	Provides the ability to customize initialization of an instance. The default definition is empty. <i>Keylist</i> is an alternating list of initialization keywords and values. See the “Initialization” section for more detail.
<code>:eql</code> <i>instance</i>	Method for comparing two instances. The exact semantics are discussed below under <i>Equality Methods</i> .
<code>:equal</code> <i>instance</i>	Method for comparing two instances. The exact semantics are discussed below under <i>Equality Methods</i> .

<code>:equalp</code> <i>instance</i>	Method for comparing two instances. The exact semantics are discussed below under <i>Equality Methods</i> .
<code>:typep</code> <i>type</i>	This method returns T if <code>self</code> is of instance type <i>type</i> , and nil otherwise.
<code>:copy</code>	This method is used to make a copy of an object. The default definition simply returns the instance the method was invoked on.
<code>:copy-instance</code>	This method returns a new instance of the same type as <code>self</code> whose instance variables denote the same objects as the corresponding instance variables of <code>self</code> .
<code>:copy-state</code>	This method returns <code>self</code> . It is present so that it can be redefined to implement a copy method for a user-defined instance type.

## Equality Methods

There are three universal methods for comparing two instances for equality: `:eql`, `:equal`, and `:equalp`. The default definitions of these methods are:

```
(define-method (the-type :eql) (another-instance)
  (eq self another-instance))

(define-method (the-type :equal) (another-instance)
  (call-method :eql another-instance))

(define-method (the-type :equalp) (another-instance)
  (call-method :equal another-instance))
```

By default, all three methods return T only if the two instances being compared are the same object. Notice though that if you change the definition of `:eql`, you are changing the meaning of `:equal` and `:equalp`. Similarly, if you redefine the `:equal` method, then `:equalp` will work differently.

If you redefine any of the equality methods, your definition should be

1. Reflexive.  $(\Rightarrow \text{foo equality-method foo}) \Rightarrow \text{True}$
2. Symmetric. If  $(\Rightarrow \text{foo1 equality-method foo2}) \Rightarrow \text{True}$ , then  $(\Rightarrow \text{foo2 equality-method foo1}) \Rightarrow \text{True}$ .
3. Transitive. If  $(\Rightarrow \text{foo1 equality-method foo2}) \Rightarrow \text{True}$  and  $(\Rightarrow \text{foo2 equality-method foo3}) \Rightarrow \text{True}$ , then  $(\Rightarrow \text{foo1 equality-method foo3}) \Rightarrow \text{True}$ .

The definitions of the Common Lisp functions `eq1`, `equal`, and `equalp` have been extended to allow them to compare instances. If only one of the arguments to these functions is an instance, then `nil` is returned. If both of the arguments are instances, then the corresponding equality method is invoked on the first instance with the second one as its argument. i.e.,

```
(equal foo1 foo2)
```

where `foo1` and `foo2` are instances, becomes

```
(send? foo1 :equal foo2)
```

Note the use of `send?` to avoid an error if there is no `:equal` method defined for `foo1`.

## Checking the Type of Instances

The default definition of the universal method `:typep` takes one argument and returns `T` if the argument is the same as the name given in the `define-type`.

The definition of the Common Lisp function `typep` has been extended to allow it to check for types defined by a `define-type`. If the second argument to `typep` is a type defined by a `define-type`, then the `:typep` method is invoked on the object being checked with the type as its argument.

```
(typep x 'foo)
```

where `foo` is the name of an instance type, becomes

```
(send? x :typep 'foo)
```

A new type has also been introduced into the language: `instance`. This is a full-fledged type symbol whose members are any objects that are instances. The form

```
(typep x 'instance)
```

will return `T` for any `x` that is an instance, and `nil` for any object that is not.

The function `instancep` has also been added to the language.

```
(instancep object)
```

*Function*

This returns `T` whenever `object` is an instance. Otherwise it returns `nil`.

The Common Lisp function `type-of` has been extended to return the name of the instance type of its argument whenever that argument is an instance.

## Copying Instances

The `:copy-instance` and `:copy-state` universal methods are intended to be used as part of the implementation of a user-defined copy method on a user-defined instance type. This will typically be done by invoking the `:copy-instance` method on the instance to be copied, and then invoking a user-defined `:copy-state` on the resulting instance to make copies of the values of instance variables.

For example,

```
(define-type murphy
  (:var x1)
  (:var x2)
  (:inherit-from cow)
)

(define-method (murphy :copy) ()
  (=> (call-method :copy-instance) :copy-state))

(define-method (murphy :copy-state) ()
  (call-method (cow :copy-state))
  (setq x1 (=> x1 :copy))
  (setq x2 (=> x2 :copy))
  self
)
```

This example assumes that the values of the instance variables `x1` and `x2` are instances that have a `:copy` method, and that the instance type `cow` has a similar user-defined `:copy-state` method. For instance variables whose values are not guaranteed to be instances, you need to consider each possible type of object that they could be.

---

## Redefining Instance Types

While developing a program, you may need to change the definition of an instance type. This can have serious implications, since there may be objects of that type already in the system, or other instance types that inherit from the type you're changing. If the new definition is compatible with the old (old methods will work correctly on new objects, and new methods will work on old objects), then the system will let you change the definition of the instance type. If the new definition is not compatible with the old, then an error is signalled and you must either rename the existing instance type, or abort the new definition.

The function `rename-type` is used to change the name of an existing instance type.

`(rename-type symbol1 symbol2)`

*Function*

If there is an existing instance type *symbol1* and no instance type *symbol2*, then the type identified by *symbol1* is renamed to *symbol2*, otherwise an error is signalled. A call to `rename-type` returns the symbol that names the new type.

Assume that before the `rename-type`, *symbol1* was the name of type *a*. Existing objects of type *a* will continue to function, and will be affected by changes made to *a* using the name *symbol2*. Calling `type-of` with one of these "old" objects will return *symbol2*. A new instance type named *symbol1* can be defined without affecting any of the type *a* instances.

If an existing type *child* inherits from type *a*, it will continue to refer to type *a*, and will be affected by any subsequent changes made to *a* using *symbol2*. This is true until the type *child* itself is redefined, at which point the names in its `:inherit-from` option(s) are reinterpreted.

Compiled or preprocessed invocations of `=>` where the target variable has been declared to be type *a* will work only on instances of type *a*, even if a new type with the name *symbol1* is defined.

## Undefining Instance Types

An instance type can also be removed entirely from the system. This of course makes useless any existing instances of that type. Attempting to send a message to an instance whose type has been undefined results in an error.

`(undefine-type symbol)`

*Function*

If there is no instance type named *symbol*, `undefine-type` returns `nil`. Otherwise, the type *a* named by *symbol* is removed from the system, and `T` is returned.

If there is any existing type that inherits from *a*, an error is signalled and you are asked whether or not you wish to continue. If you still wish to undefine the type, continue the computation with the NMODE command `lisp-C` (or the listener macro `!c`, which calls `sys:listener-continue`). If you undefine an instance type that is inherited by other types, the other types will most likely be rendered useless also, since they probably use methods defined on type *a*.

You can also undefine a particular method for an instance type.

`(undefine-method instance-type method-name)`

*Function*

A call to the function `undefine-method` removes the existing definition of *method-name* for *instance-type*. An error is signalled if there is no instance type *instance-type*. If there is no method *method-name* for *instance-type*, no error is signalled, but `nil` is returned.

# Notes



# Calling Non-Lisp Routines

---

## Introduction

Hewlett-Packard has enhanced Common Lisp by providing the ability to call routines written in other programming languages (e.g., C, Fortran, Pascal) from Lisp. Routines written in another language are sometimes called **foreign functions** or **non-Lisp functions**. Presently this capability works only in one direction: you can call a foreign function from Lisp, but there is no way to call a Lisp function from a program written in another language.

This chapter explains how to call a foreign function from Lisp; it has the following organization:

- Defines background concepts (object files and entry points).
- Describes how to load conventional object code into your system.
- Shows how to create a Lisp function that calls a loaded foreign function (includes parameter conversion and how to access a foreign global variable).
- Examples for each of C, Pascal, Fortran, and assembly language.

Note that all of the symbols that name the functions and variables described here are in the `extn` package. The `external` module must be loaded before the non-Lisp routine calling facility can be used.

```
(require "external")
```



---

## Background Information

Here is some information about the HP-UX implementation of the three conventional languages (C, Fortran, and Pascal) that you can call from Lisp. Knowing this information makes it easier to understand the mechanism used to access foreign functions from Lisp.

### Object File Format

The C, Pascal and Fortran compilers that run on HP-UX produce relocatable object files that have a common format. These are known as `.o` (pronounced “dot o”) files because this is the default suffix for files of this type. The exact format of a `.o` file is described in the *a.out(5)* entry of the *HP-UX Reference*.


Object files may contain references to things that are not defined within that file. These are called **external references**, and they must be resolved when the object files are loaded. Typically the external “thing” is defined in another object file that is being loaded at the same time, or in an archive (library), in which case the necessary part of the archive is linked and loaded along with the object file.

### Entry Points

The names within an object file or code archive that are available to resolve external references in other object files are known as **entry points**. Different compilers have different ways of constructing the name of an entry point from the name of the corresponding entity in the source code. The Series 300 C compiler simply adds an underscore to create the name of the entry point, so in the object file for a C function named `goof`, there is an entry point `_goof`. The compiler conventions for naming entry points in Pascal and Fortran are discussed under the appropriate examples later in this chapter. You can use the HP-UX command `nm(1)` to identify the entry points in an object file.

---

## Loading Foreign Functions




The code that supports the foreign function call mechanism is in the `external` module. To call foreign functions, this module must be loaded. If it wasn't loaded when your system was created, you must load the module by executing

```
(require "external")
```

Before calling a foreign function, the object file that contains the definition for the function must be loaded into your Lisp environment. This is done with `extn:load-ofile`.


```
(extn:load-ofile object-file-names &key :libs :protecting :redefining      Function  
              :entry-point-format :load-also :verbose :print)
```

The parameter *object-file-names* is a filename or list of filenames, identifying the relocatable object file(s) to be loaded into the system. Typically these are `.o` files that were created by compiling a source file with the `-c` (suppress link edit phase) option. A filename is either a string or a pathname. If *object-file-names* is `nil`, then no `.o` files are loaded; this is used in combination with the `:load-also` parameter to load library routines.



The `:libs` keyword parameter is a string or list of strings specifying the libraries to be loaded. A library in `/lib` or `/usr/lib` can be specified with `"-library-name"`; any other library is specified with its complete HP-UX path. If no `:libs` parameter is provided, the value of `*default-hpux-libraries*` (initially `"-lc"`) is used. Note that when the `:libs` parameter is provided, the C library is not used to resolve external references unless it is specified in that parameter.

The keyword parameter `:redefining` has to do with external references and the way object files are loaded into the system. External references in the loaded object files are satisfied first by any entry points already defined in the Lisp environment, and then by the libraries specified by the `:libs` parameter. If you wish to override this by having an entry point already in the system replaced by one with the same name from the libraries you are loading, you must specify the name(s) of the entry point(s) with the `:redefining` parameter. A single entry point is specified as a string or symbol; multiple entry points are specified as a list of strings and/or symbols.



On the other hand, the keyword parameter `:protecting` identifies entry points present in the system that should **not** be redefined by the call to `load-ofile`, even if there is an entry point of the same name defined in the file(s) being loaded. A single entry point is specified as a string or symbol; multiple entry points are specified as a list of strings and/or symbols.

The keyword parameter `:load-also` specifies entry points that should be loaded from the libraries even if no external references to them exist in the files being loaded. This is especially useful if you want to access only some library routines. A single entry point is specified as a string or symbol; multiple entry points are specified as a list of strings and/or symbols. The value of `:load-also` defaults to `nil`.

The actual names of entry points that are affected by the `:redefining`, `:protecting`, and `:load-also` keywords are constructed by passing each specified entry point name (a string or symbol) and the `:entry-point-format` parameter (defaults to `*default-entry-point-format*`) to `format`.

The keyword parameters `:verbose` and `:print` are similar to those for the Common Lisp function `load`.

The Common Lisp function `load` has been extended to allow its argument to be the name of a `.o` file. Essentially,

```
(load "fingle.o") ≡ (extn:load-ofile "fingle.o")
```

#### Examples:

```
(require "external")
(use-package 'extn)

(load-ofile "/users/joe/clock.o")           ; Load object from specific file
(load-ofile nil :libs "-lc" :load-also      ; Load some C library functions
  '(tmpnam
    "tmpnam")
)
(load-ofile "windows.o" :libs '("-lwindow" "-lsb1" "-lsb2" "-lc"))
```

## Load-related Variables and Functions

There are some system variables that affect the function `load-ofile`. The symbols that name these variables are in the `extn` package.

`extn:*unexported-entry-points*`

*Variable*

The value of `*unexported-entry-points*` is a list of entry points that may be present in the system, but are not used to satisfy external references in loaded `.o` files or libraries. The entry points initially on the list are `_end`, `_etext`, and `_edata`. To construct an entry point to be added to `*unexported-entry-points*`, use the function `entry-point-symbol` described below.

`extn:*protected-entry-points*`

*Variable*

The value of `*protected-entry-points*` is a list of entry points that cannot be redefined by loading a `.o` file or library, even if the files being loaded contain an entry point with that name and the entry point is specified by the `:redefining` parameter. The entry points initially on the list are `_end`, `_etext`, and `_edata`. To construct an entry point to be added to `*protected-entry-points*`, use the function `entry-point-symbol` described below.

Since the entry points on these lists are maintained as symbols instead of strings, a function is provided to construct an appropriate symbol.

`(extn:entry-point-symbol name &key :entry-point-format)`

*Function*

This function returns a Lisp symbol that identifies the entry point with the name *name* (a string) that has been loaded into the system with `load-ofile`. If no entry point with that name has been loaded, `nil` is returned. The name of the symbol is constructed by passing *name* and the `:entry-point-format` parameter to `format`. The default for `:entry-point-format` is the value of `*default-entry-point-format*`.

`extn:*default-entry-point-format*`

*Variable*

The value of `*default-entry-point-format*` is used as the default value of the `:entry-point-format` keyword parameter for the functions `load-ofile`, `entry-point-symbol`, `define-entry-point`, `defexternal`, and `defexternalvar`. It should be a string suitable for use as the format string in a call to `format`. The initial value of `*default-entry-point-format*` is `"~(_A~)"`. (This says stick an underscore at the beginning of the argument and make it lowercase.)

It is possible that an object file you are loading contains an external reference that you know is never used by the function you want to access from Lisp. In this case you would not want the loader to resolve this reference by linking in a library function that would just waste space. For this reason, a function is provided to define a “dummy” entry point that will be used to resolve the external reference at load time.

`(extn:define-entry-point name &key :entry-point-format :value)` *Function*

The name of the entry point is obtained by calling `format` with `nil`, the value of `:entry-point-format` (defaults to `*default-entry-point-format*`), and `name`. If the entry point already is present in the system, `define-entry-point` returns `nil`, otherwise, `T`.

The `:value` keyword parameter can be used to assign a value to the entry point. This is only useful if you are familiar with the implementation of HP-UX and HP's Common Lisp. The default for `:value` is a value that will signal an error (such as a bus error) if it is ever used for anything besides satisfying an unused external reference.

---

## Creating an Access Routine

Once you have loaded the object code that contains the entry point for the function you wish to call, you must create a lisp macro or function to call the foreign function. The macro `extn:defexternal` defines the calling macro or function for you.


`(extn:defexternal func-name param-list &key :result :entry-point  
:entry-point-format :macro :type-check)` *Macro*

*Func-name* is the name of the function or macro being defined (as in `defun` or `defmacro`).

*Param-list* is a list of the parameter specifications for the function or macro being defined. It indicates the number and types of parameters to the foreign function as well as what types of Lisp objects are expected as parameters to the function or macro being created. The syntax for parameter specifiers is explained below under "Parameter Specifiers". If the function has no parameters, *param-list* should be `nil`.

The keyword parameter `:result` indicates the type of result the non-Lisp function will be returning and what Lisp type it should be converted to. The syntax for this result specifier is discussed below under "Result Specifiers". If `:result` is omitted, the function or macro created will return `nil`.

The keyword parameters `:entry-point` and `:entry-point-format` identify the entry point to be called by the macro or function being defined. The parameter `:entry-point` defaults to *func-name*, and `:entry-point-format` defaults to the value of `*default-entry-point-format*`. The name of the entry point is obtained by calling `format` with `nil` and these two parameters. While `:entry-point-format` is evaluated, `:entry-point` is not.




The keyword parameter `:macro` indicates whether to create a function or a macro. If `nil` (the default), a function is created, otherwise a macro is created. It may be desirable to create a macro to avoid the overhead of a Lisp function call, but care should be taken when using macros created by `defexternal` so that expressions passed in as arguments have no side effects and do not allocate space (cons).

If the keyword parameter `:type-check` is true (`nil` is the default), the function created by `defexternal` will do run time type checking of array parameters.

## Parameter Specifiers

Since Lisp uses different representations for data than do the other languages available on HP-UX, the arguments to a function created by `defexternal` must be converted before they are passed to the foreign function. Necessary conversions are done automatically according to the *param-list* parameter to `defexternal`. The syntax of these specifiers will be described shortly.


### Non-Array Parameters



There are two styles of parameter specifiers for parameters that are not arrays. Which one you use depends on whether the argument for that parameter is passed by reference and whether the non-Lisp code could change the value of the argument.

Any arguments of non-Lisp functions that are passed by reference (i.e., a pointer to the argument is passed instead of its value) require special attention. Many Lisp objects (such as numbers) are considered immutable and are not expected to change. Passing a pointer to the representation of an immutable object to a non-Lisp function gives that function the ability to change the object. This could cause problems if the object is used by other Lisp constructs. It is considered an error to directly pass pointers to immutable Lisp objects to non-Lisp code unless that code is known to not modify the object. It is not considered an error to pass pointers to mutable objects such as strings.

The arguments corresponding to the following types of non-Lisp parameters are passed as pointers:

- All parameters of Fortran functions
  - Pascal var parameters
  - Pascal parameters declared to be pointers (eg. `i: ^integer`)
  - C parameters declared to be pointers (eg. `int *i`)
- 

This restriction does not make it impossible to use non-Lisp code that modifies its arguments, but it does require you to use the correct style of parameter specifier when setting up the interface to the non-Lisp function.

The first style of parameter specifier is used for non-pointer parameters and for pointer parameters that you are certain will not be modified by the non-Lisp routine. The specifier is a list whose first element indicates the type of the Lisp object to be passed and whose second element indicates the non-Lisp type it is to be converted to.

*(lisp-type non-Lisp-type)*

If the non-Lisp type is a pointer type, the non-Lisp code should not modify the value of the parameter. An example of this style of parameter specifier is `(fixnum p-integer)`, which specifies passing a Lisp fixnum to a Pascal function expecting a non-var integer argument.

The second style of parameter specifier is used for specifying parameters whose corresponding arguments may be changed by the non-Lisp code. It has the form:

*(var lisp-type non-lisp-type)*

In this case the Lisp caller must pass a symbol whose global value is of type *lisp-type*. Depending on *Lisp-type*, the non-Lisp function is either passed a pointer to the value or a copy of the value. After the non-Lisp call, the global value of the symbol will be set to the (possibly modified) value of the argument. This may be the original object mutated, or an entirely new object. This is the recommended way of passing by reference (pointer) when the argument may be modified.

A parameter specifier that is `nil` indicates that no conversion is to be performed on that parameter.

The following table shows the symbols that are currently supported for use in non-array parameter specifiers. In the names, \* indicates a C pointer, and ^ indicates a Pascal pointer (or var parameter). These names are symbols in the extn package. The letters next to the non-Lisp types indicate what Lisp types can be converted to that non-Lisp type.

Lisp	C	Pascal	Fortran
integer	c-int	p-boolean	f-integer
fixnum	c-char	p-char	f-short-integer
float	c-short	p-scalar	f-real
simple-string	c-long	p-integer	f-double
string	c-unsigned-int	p-real	f-logical
character	c-unsigned-char	p-longreal	f-short-logical
string-char	c-unsigned-short	p-string	f-character
boolean	c-unsigned-long	p-pac	f-Hollerith
	c-float		
	c-double	p-^boolean	
		p-^char	
	c-int*	p-^scalar	
	c-char*	p-^integer	
	c-short*	p-^real	
	c-long*	p-^longreal	
	c-unsigned-int*	p-^string	
	c-unsigned-char*	p-^pac	
	c-unsigned-short*		
	c-unsigned-long*	p-varstring	
	c-float*		
	c-double*		
	c-string		

Key to the letter codes:

A: integer fixnum simple-string string character string-char boolean

B: integer fixnum simple-string string boolean

C: integer fixnum character string-char boolean

D: integer fixnum boolean

E: simple-string string

F: float

G: simple-string (can only be used in a var specifier)



A few clarifications of the meaning of some non-Lisp type specifiers:

- The Lisp type `boolean` refers to values whose only significance is whether they are `nil` or `non-nil`.
- The specifier `p-varstring` represents a Pascal var parameter declared as `string` rather than as `string[n]`.
- The specifier `p-pac` represents a Pascal packed array of characters.

### Array Parameters

You can also pass simple arrays to non-Lisp functions. Arrays are passed as pointers, so any changes made to an array by a non-Lisp routine will be reflected in the array after the routine returns.

The syntax for specifying an array parameter to a non-Lisp function is

```
(({vector | array} lisp-array-type) non-lisp-array-type)
```

If you're passing a one-dimensional array, use `vector`, otherwise use `array`.

*Lisp-array-type* indicates the type of the elements in the arrays to be passed. It can have any of the following values.

```
bit string-char character fixnum integer float T
```

If *lisp-array-type* is `T`, it says that the array or vector is a general Lisp array or vector. The types of the elements of the array must be convertible to the element type of the non-Lisp routine's array. All of the Lisp array's elements must be the same type. Specifying *lisp-array-type* to be `T` is less efficient than giving a specific type.

---

#### NOTE

The current implementation does not use any special representation for arrays with element type `character`, `fixnum`, or `integer`. Consequently, using one of these types as the *lisp-array-type* is the same as using `T`. This behavior may change in future releases.

---

If *lisp-array-type* is not T, then the array must have been created by a call to *make-array* with the proper *:element-type* keyword parameter. Likewise, if the parameter was specified as being a vector, then a vector must be passed. The default is for no type checking of array parameters to non-Lisp routines, but you can have the routine created by *defexternal* check for these cases by proclaiming *safety* to be 3 before the *defexternal* with `(proclaim '(optimize (safety 3)))`, or with the *:type-check* parameter to *defexternal*. **Passing arrays of the wrong type when type checking is not enabled can be hazardous to the state of your system.**

The following table shows valid values for *non-lisp-array-type* along with codes designating which Lisp array types can be converted to them.

C		Fortran		Pascal	
<code>c-char*</code>	A	<code>f-character-array</code>	A	<code>p-pac</code>	A
<code>c-int*</code>	B	<code>f-integer-array</code>	B	<code>p-integer-array</code>	B
<code>c-long*</code>	B	<code>f-short-integer-array</code>	B	<code>p-real-array</code>	C
<code>c-float*</code>	C	<code>f-real-array</code>	C	<code>p-longreal-array</code>	C
<code>c-double*</code>	C	<code>f-double-array</code>	C	<code>p-char-array</code>	A
<code>c-*</code>	D				

- A: string-char character T
- B: character fixnum integer T
- C: float T
- D: anything

Note that although T is syntactically correct with every non-Lisp array type, the elements of the array still may need to be converted. For an idea of the conversions that are available, see the table in the section "Non-Array Parameters" above.

Multi-dimensional Fortran arrays are stored in column-major order while Lisp arrays are stored in row-major order (as are C and Pascal arrays). This means that the indices of Lisp arrays must be transposed when they are accessed by Fortran routines. There is an example of this in the section below containing Fortran examples.

## Result Specifiers

Like parameter specifiers, result specifiers are represented as lists, but the first element is the non-Lisp type of the result, and the second element is the Lisp type the result should be converted to, i.e.,

*(non-Lisp-type Lisp-type)*

The possible type names are the same ones that are available for parameter specifiers (see above).

The result specification `null` indicates that the function created by `defexternal` should return `nil`. The result specification `nil` says not to perform any conversion of the result.

## Restrictions

There are a few reasonable restrictions to the non-Lisp function calling facility.

- No type checking is performed on the arguments to a function or macro created by `defexternal`, except for array parameters when `safety` is proclaimed to be 3 or the `:type-check` keyword is specified to be true.
- There is no way to obtain a useful value from a C function that returns a structure by value (eg. `struct stype sfunc()`).
- Records, structures, sets, and other “complicated” types are not supported.
- Lisp strings cannot be passed as arguments to Pascal functions whose corresponding parameter is declared to be a string of length less than four.

## Examples

Here are some examples of calls to `defexternal`. These assume that the necessary entry points have already been defined by a prior call to `load-ofile`. Complete examples showing every step needed to call functions in C, Fortran, and Pascal, are given at the end of this chapter.

```
(require "external")
(use-package 'extn)

;;; Define C_FUNC_NUMBERS to call the entry point "_c_func_numbers".
;;; This C routine takes six arguments and returns a double float
(defexternal c_func_numbers ((integer c-short)
                             (integer c-int)
                             (integer c-long)
                             (integer c-unsigned-int)
                             (float c-float)
                             (float c-double))
                          :result (c-double float))

(defexternal clock nil :result (c-long integer))
```

---

## Accessing Non-Lisp Variables

In addition to being able to define Lisp functions that call non-Lisp functions, it is also possible to create a Lisp function to return the value of a global variable that is defined in a program written in another language.

```
(extn:deffunction function-name &key :vartype :entry-point Macro  
                :entry-point-format :offset :macro)
```

This macro is used to create functions and macros of no arguments that return values from non-Lisp global variables. The non-Lisp routines must be loaded (by a call to `load-ofile`) before calling the created function or macro. The `:entry-point`, `:entry-point-format`, `:offset`, and `:macro` parameters are evaluated, but the other parameters are not.

The name of the Lisp function or macro created is the symbol *function-name*. The keyword parameters are as follows:

- |                                  |   |
|----------------------------------|---|
| <code>:vartype</code>            | Specifies the conversion of the non-Lisp value to a Lisp value. These have the same form and possibilities as result specifiers for <code>deffunction</code> (except <code>null</code> is not allowed). A <code>:vartype</code> of <code>nil</code> indicates that no conversion is to be performed. It defaults to <code>(c-int integer)</code> .  |
| <code>:entry-point</code>        | This identifies the entry point associated with the variable. The value of <code>:entry-point</code> defaults to <i>function-name</i> . The actual entry point name to be used is obtained by calling <code>format</code> with <code>nil</code> and the values of the <code>:entry-point</code> and <code>:entry-point-format</code> keyword parameters.  |
| <code>:entry-point-format</code> | Used to construct the name of the entry point to be accessed. It defaults to the value of <code>*default-entry-point-format*</code> . See <code>:entry-point</code> .   |
| <code>:offset</code>             | Used to reference variables at a byte offset from the named entry point. This is useful when referencing globals in Pascal and Fortran where globals are maintained as offsets into a global or common area. It is also useful for accessing elements of global records, structures, or arrays. User supplied offsets must be derived from a knowledge of how the non-Lisp compilers allocate memory for global variables. This argument defaults to 0. |
| <code>:macro</code>              | Specifies whether a function or macro is to be created. A value of <code>nil</code> (the default) means a function is created; non- <code>nil</code> says to create a macro. Creating a macro avoids the overhead of a function call, but if complicated type conversions are required, the advantage may be negligible or nonexistent.   |

## Example

As an example of where `defexternalvar` is useful, consider the HP-UX system variable `errno`. If you are making an HP-UX system call from Lisp (with a function created by `defexternal` or with one of the provided functions) and something goes wrong, it would be nice to be able to check what the specific error was. The following code defines a function `hp-ux-error` that returns the value of `errno`. (For more information on `errno` see *errno(2)* in the *HP-UX Reference*.)

```
(require "external")
(use-package 'extn)

;;;
;;; Load the errno entry point from the C library.
;;;
(load-ofile nil :load-also "errno")

;;;
;;; Define hp-ux-error with defexternalvar
;;;
(defexternalvar hp-ux-error :entry-point "errno")
```

---

## Complete Examples

This section presents examples (at least one each for C, Pascal, Fortran, and assembly language) of calling non-Lisp functions from Lisp. Every step in the process is described. Even if you don't need to call functions written in all three of these languages, you may want to look at all of the examples for a more complete understanding of the non-Lisp function calling mechanism.

### C

Suppose you have the following C function that you wish to call from Lisp defined in the file `silly.c`.

```
double silly(x1, x2, name)
    int x1;
    float *x2;
    char *name;

{
    if ( strcmp(name, "Big Cheese") == 0) {
        *x2 = 8.762;
        return(x1 + *x2);
    }
    else
        return(x1 - *x2);
}
```

The first step is to compile the source file into a `.o` file with the HP-UX command

```
cc -c silly.c
```

This will produce the object file `silly.o` containing the entry point `_silly`. Entry points in C are just the name of the function or global variable preceded by an underscore (`_`). These entry point names are case sensitive.

Assuming that the file `silly.o` is in the directory `$HOME/c-code`, the Lisp code to access this function looks like this:

```
(require "external")
(use-package 'extn)

;;;
;;; Load the .o file containing the object code for silly
;;;
(load-ofile "$HOME/c-code/silly.o")

;;;
;;; Define a function lisp-silly that calls the C function silly
;;;
(defexternal lisp-silly ((fixnum c-int)
                        (var float c-float*) ; Argument must be symbol
                        (simple-string c-char*))
  :result (c-double float)
  :entry-point "silly") ; Note use of default format

(setq a2 3.221)
;;;
;;; Call lisp-silly
;;;
(let ((a1 5) (a3 "Big Cheese"))
  (lisp-silly a1 'a2 a3) ; Note that a2 is quoted (var param).
)

a2 ⇒ 8.762
```

This next example shows calling a C function that has an array parameter. Assume that the C source is in a file called `linear.c`.

```
/*  
 Finds the determinant of a 2x2 integer array.  
*/  
determinant(a)  
int a[2][2];  
{  
    return((a[0][0] * a[1][1]) - (a[0][1] * a[1][0]));  
}
```

After you have compiled this into the file `linear.o` with the HP-UX command

```
cc -c linear.c
```

you can access the function `determinant` from Lisp.

```
(require "external"  
(use-package 'extn)  
  
(load-ofile "linear.o")  
  
(defexternal determinant (((array fixnum) c-int*))  
    :result (c-int fixnum))  
  
(setq m (make-array '(2 2) :initial-contents '((-1 9) (-8 10))  
    :element-type 'fixnum))  
  
(determinant m) ⇒ 62
```



The following example shows the Lisp code to call some HP-UX library routines from Lisp. See section 3 of the *HP-UX Reference* for details about the routines.

```
(require "external")
(use-package 'extn)
;;;
;;; Load the necessary object code from the C library
;;;
(load-ofile nil :libs "-lc" :load-also '("clock" ; "-lc" is unnecessary
                                         "atoi"))

;;;
;;; Create the Lisp functions to call the C library routines
;;;
(defexternal clock nil :result (c-long integer))

(defexternal atoi ((simple-string c-string))
                  :result (c-int integer))

;;;
;;; Now call the lisp functions
;;;
(clock)

(atoi "456") ⇒ 456
```

## Pascal

Calling Pascal functions from Lisp is similar to calling C functions, except for the way the Pascal compiler identifies entry points. This example illustrates the differences.

Given the following Pascal source code in a file `ipswich.p`,

```
module foo;

export    type mystring = string[15];
          function pfunc1(x1:integer; var x2:real; name:mystring): real;
          function pfunc2(n: integer; x, y:real): longreal;

implement

function pfunc1;
begin
    if (name = 'Big Cheese') then begin
        x2:= 8.762;
        pfunc1:= x1 + x2
    end
    else
        pfunc1:= x1 - x2
    end;

function pfunc2;
var
    i: integer;
    temp: longreal;
begin
    temp:= 0;
    for i:= 1 to n do
        temp:= temp + x + y;
    pfunc2:= temp
end;

end. { MODULE FOO }
```

To compile this module of two functions, execute the HP-UX command

```
pc -c ipswich.p
```

This will create the file `ipswich.o`.

The Pascal compiler has a different convention for naming entry points in the object files it creates. Entry points in Pascal are all converted to lower case. The form of the entry point name is *\_module-name\_routine-name* or *\_program-name\_routine-name* if you're not using modules. This affects how we identify the entry points in calls to `defexternal` and `load-ofile`.

```
(require "external")
(use-package 'extn)
;;;
;;; Load the necessary object code
;;;
(load-ofile "ipswich.o" :libs '("-lpc -lc"))

;;; Set up an entry point format string for module foo
(defvar foo-entry-point-format "_foo_~(A~)")

;;;
;;; Define the Lisp equivalents of pfunc1 and pfunc2
;;;
(defexternal pfunc1 ((fixnum p-integer)
                    (var float p~real) ; Argument must be a symbol
                    (string p-string))
  :result (p-real float)
  :entry-point-format foo-entry-point-format)

(defexternal lisp-pfunc2 ((fixnum p-integer)
                        (float p-real)
                        (float p-real))
  :result (p-longreal float)
  :entry-point "pfunc2"
  :entry-point-format foo-entry-point-format)

;;;
;;; Call the functions
;;;
(setq p-arg 4.566)
(pfunc1 2 'p-arg "Jack be nimble") => -2.566

(lisp-pfunc2 11 8.9 7.3) => 178.2
```

## Procedures

Another difference between Pascal and C is that Pascal can have procedures as well as functions. This next example demonstrates calling a Pascal procedure with an array parameter from Lisp.

```
module hunky;

export type matrix = array[1..3,1..3] of integer;
      procedure dory(var m:matrix);

implement

procedure dory(var m:matrix);

var   i,j: integer;
      temp: matrix;
begin
  for i:= 1 to 3 do
    for j:= 1 to 3 do
      temp[i,j]:= m[j,i];
    for i:= 1 to 3 do
      for j:= 1 to 3 do
        m[i,j]:= temp[i,j]
      end;
    end. { MODULE HUNKY }
```

Compile this function (defined in the file pproc.p) with

```
pc -c pproc.p
```

to create the file pproc.o. In this case the entry point for the procedure is named `_hunky_dory`.

```

(require "external")
(use-package 'extn)

;;;
;;; Load the necessary object code
;;;
(load-ofile "pproc.o" :libs '("-lpc" "-lm" "-l")

;;;
;;; Define the Lisp equivalent of procedure dory
;;;
(defexternal dory (((array fixnum) p-integer-array))
  :result null ; Always return nil
  :entry-point-format "_hunky_~(^A~)"
  :type-check t)

;;;
;;; Call the function
;;;
(setq my-matrix (make-array '(3 3) :initial-contents
  '((1 2 3) (4 5 6) (7 8 9))))

(dory my-matrix) ⇒ NIL
my-matrix ⇒ #2A((1 4 7) (2 5 8) (3 6 9))

```

## Fortran

The important thing to remember when writing Lisp code to access Fortran functions is that all parameters are passed by reference. Thus if an argument's value may be changed by the Fortran function, that corresponding parameter should be specified using `(var Lisp-type non-Lisp-type)` in the `defexternal` call.

Suppose you wanted to call the following Fortran function (defined in the file `archaic.f`).

```
real function oof(x1, x2, name)
integer x1
real x2
character*15 name

if (name .eq. 'Big Cheese') then
    x2 = 8.1
    oof = x1 + x2
else
    oof = x1 - x2
end if

end
```

Compiling this with

```
fc -c archaic.f
```

creates the object file `archaic.o`. The Fortran compiler names entry points like the C compiler (i.e., `_function-name`).

After the file is compiled, the following Lisp code can be used to call the function `oof`.

```
(require "external")
(use-package 'extn)
;;;
;;; Load the required object file
;;;
(load-ofile "archaic.o" :libs '("-lF77" "-lI77" "-lm" "-lc"))

;;;
;;; Set up the equivalent Lisp function
;;;
(defexternal oof ((fixnum f-integer) ; This parameter isn't changed
                 (var float f-real) ; This one could be changed
                 (string f-character))
                 :result (f-real float))

(setq fparam 2.33)
;;;
;;; Call the created function
;;;
(oof 0 'fparam "Big Cheese" ) ; Note padding to Fortran size (*15)
fparam => 8.1
```

## Subroutines

Fortran also has program units that are not functions. This example shows calling a Fortran subroutine that has array parameters. Remember that Fortran stores arrays in column-major order while Lisp stores them in row-major order. This means that you need to reverse the subscripts.

```
*      Puts the sum of the rows of array x into array sum
      subroutine jawaka (x, sum)
      integer x(3,5),sum(3)

      do 100 i=1,3
        sum(i) = 0
        do 200 j=1,5
          sum(i) = sum(i) + x(i,j)
200      continue
100     continue
      end
```

Assuming this is defined in the file `fproc.f`, compile it into the file `fproc.o` with

```
fc -c fproc.f
```

Then, you could use the following Lisp code to access the subroutine.

```
(require "external")
(use-package 'extn)
;;;
;;; Load the required object file
;;;
(load-ofile "fproc.o" :libs "-lF77 -lI77 -lm -lc")

;;;
;;; Set up the equivalent Lisp function
;;;
(defexternal lisp-jawaka
  (((array t) f-integer-array)
   ((vector t) f-integer-array)
  )
  :entry-point "jawaka"
  :result null      ; Always return nil
)

;;; Call the created function
(setq mtrx (make-array '(5 3) ; Note the transposition of dimensions
  :initial-contents '((1 2 3) (4 5 6) (7 8 9)
    (10 11 12) (13 14 15)))
  sumvector (make-array 3 :initial-element 0))

(lisp-jawaka mtrx sumvector)
sumvector => #(35 40 45)
```

## Assembly Language

For assembly language, entry points have no system-enforced naming conventions since they are whatever you choose. Here's an example of an assembly routine that you might inexplicably wish to call from Lisp.

```
        globl  palindromep
palindromep
* Returns 1 if parameter is a palindrome
* (i.e., reads same forwards and backwards)
* Returns 0 if not.
* Comparison is case sensitive, all chars examined.
        move.l 4(a7),a5      Get char pointer s
        move.l a5,a4        Copy it to p
LOOP1   tst.b  (a4)          Bump p until it points at 0
        jeq   END1
        addq.w #1,a4
        jra  LOOP1
END1    subq.w #1,a4        Back up p to point at last char

LOOP2   cmp.l  a4,a5        Loop until s >= p
        jcc  END2          or until *s != *p
        move.b (a5),d0
        cmp.b (a4),d0
        jne  END2
        addq.w #1,a5        s++
        subq.w #1,a4        p--
        jra  LOOP2        Go to top of loop

END2    moveq  #0,d0        Return 1 if palindrome, 0 if not
        cmp.l  a4,a5
        jcs  EXIT
        moveq  #1,d0
EXIT    rts
```

If this code was in the file `aproc.s`, you would assemble it with the command

```
as aproc.s
```

to produce the file `aproc.o`. The following code could then be used to access it from Lisp.

```
(require "external")
(use-package 'extn)

(load-ofile "aproc.o")

(defexternal palindromep ((simple-string c-string))
  :result (c-int boolean)
  :entry-point-format "~(~A~)")

(palindromep "aba")           => T
(palindromep "aaa")           => T
(palindromep "abcd")         => NIL
(palindromep "abcdefgfedcba") => T
```



# Notes



# Debugging Tools

---

## Introduction

The Hewlett-Packard Lisp workstation provides tools to help you debug Lisp programs. There are two levels of access to these tools:

- A functional level where you obtain information by calling particular debugging functions.
- A higher level integrated into the human interface. This is easier to use, and likely the one you will use more often.

At the the lower level there are functions that:

- Return information about the state of a function call.
- Cause calls to a particular function to be traced.
- Set break points at particular functions.
- Signal and process error conditions.
- Inspect Lisp objects.

The two screen-oriented debugging tools are:

- An execution monitor that allows you to step through the execution of a Lisp form.
- An execution stack browser that displays the current state of the active Lisp execution environment.

This chapter discusses how to use these debugging tools, as well as briefly covering some concepts essential to understanding them.

---

## Concepts

In order to use the debugging tools effectively, you must have a little knowledge about the way things are represented in the Lisp system. For instance, you should know about the execution stack and its entries, as well as what effect compiling code has on your ability to debug it.

### The Execution Stack

The Lisp system maintains in memory an **execution stack** on which it stores information about its current state. The stack can be thought of as a collection of currently active lexical environments. The environment of a particular function call stores information pertinent to that call, such as the values of parameters and local variables. This information can be very valuable when debugging, so the debugging tools allow you to view and change some of the values stored in an environment.

### Alternate Listener Modes

Listeners are discussed in more detail in the “Concepts” chapter, but this section should provide enough information for most users.

The Lisp **listener** (or read-eval-print loop) is the function that reads an expression, evaluates it, and prints the results. Most of the time this is the behavior that you want, but there are times when you want some specific additional capabilities.

If the **debug** module is loaded, and an error occurs when executing a Lisp form, evaluation of that form stops and the system enters a new listener in **debug listener** mode. This is simply another read-eval-print loop in which you can evaluate forms to examine or alter the state of the system (e.g. function definitions, values of symbols). If the **debug** module is not loaded, you will enter a simple break loop, which is another listener mode with less capabilities than debug listener mode. Depending on what the error was, you may be able to continue the original evaluation where it left off (hopefully after you have corrected the condition that caused the error). Some errors will explicitly prompt you to enter a new value to continue the execution. You can enter debug listener mode or a break loop yourself by calling the function **break**. More information on break loops and the debug listener is provided later in this chapter.

The command-oriented inspector is another specialized listener mode. It displays and accesses Lisp data objects. It too is described in greater detail later in this chapter.

## Compiled vs. Interpreted

When a Lisp function is compiled, much of the information about that function will not be available at execution time. Interpreted code however, retains most of the information that you will be interested in when debugging. For this reason, it is much better not to compile functions that you think you will have to debug.

## Optimizations

When debugging, you should turn off all optimizations by evaluating the form

```
(proclaim '(optimize (extn:eval-speed 0)))
```

This is the default; it will prevent the language preprocessor from making functional transformations to your interpreted code. If optimizations are enabled, you may find it difficult to recognize your own code because of the transformations made by the preprocessor. (See the “Programming Tips” chapter for more detail on optimizations.) The preprocessor must expand macros, however, so code that you do not recognize may be the result of a macro expansion, not an optimization. Note that `compile-file` uses the `speed` quality (not `eval-speed`) to determine what optimizations to make.

---

## The Execution Monitor

The execution monitor is an interactive debugging tool that lets you step through and control the execution of an interpretive Lisp form. With it you can run a function a step at a time, or let it run until a specified point in its execution or until you press a key. You can also modify the values of a function's arguments or return values. Once the execution monitor is loaded with

```
(require "debug-br")
```

it can be invoked with the Common Lisp macro `step`.

```
(step form)
```

*Macro*


*Form* should not be quoted. Evaluating a call to `step` puts you into the execution monitor to step through the execution of *form*. The monitor can also be invoked with the `!step` and `!qstep` listener macros or the function `debug:step-from-listener` (see the section "Debug Listener Mode" later in this chapter). To step a method, just step a call to `extn:=>` that invokes the method you want to step.

The execution monitor looks and behaves like an NMODE browser. In fact, it appears as one of the entries in the Programming Aids browser, and can be browsed into and out of. Of course, there won't be anything interesting going on there unless you have previously initiated stepping with `step`. Only one form can be stepped at any given time, but it is possible to leave the monitor's buffer, evaluate a form, and return back to the monitor. Any changes made to the values of special variables will also change them in the functions being stepped.

### Execution Monitor Items

A line displayed in the execution monitor can be one of five things:

1. Display information such as the lines at the top that describe the monitor.
2. The next form to be executed as part of the stepping. The next command you enter controls how the form will be executed.
3. A form that has been executed as part of the stepping. It is followed by `=` and the value that it returned.
4. The name of an argument to a function call that has been stepped. It is followed by `=` and its value.
5. A return value of a stepped function call.




The indentation of a line shows its relative level of nesting. When the monitor runs out of room to indent lines, it starts over on the left side and uses a new character to precede the lines. The characters used to precede lines are (in order), |, ;, :. Where a # is displayed, it indicates that that item was not printed due to the value of `debug:*debug-print-level*`. A ... indicates that the item was not displayed due to the value of `debug:*debug-print-length*`. A B in column one indicates that a breakpoint was set on the form on that line.

The execution monitor has an inverse video bar that indicates the current line of the browser. As soon as you evaluate a call to `step`, you are put into the execution monitor with the inverse bar on the form to be stepped.

## Breakpoints

In the execution monitor, breakpoints are used to interrupt execution started by the commands **Run** or **Walk**. Breakpoints are the names of functions and methods which when called or returned from, cause the monitor to stop stepping and wait for your next command. The elements of the breakpoint list denote functions or methods. Functions are denoted by the symbol that names them. Methods are denoted with the syntax


`(:method instance-type method-name+)`



Breakpoints can also be set at an arbitrary form which you are in the middle of executing. This will cause the monitor to stop at the next call after the marked form returns (i.e., when it has been executed completely).

## Commands

What commands are valid at a particular time depends on the current line of the monitor. Also, the same command may have two different meanings, depending on the state of the monitor. This section describes the execution monitor commands, grouped by functionality.



## General Commands


These commands are always available in the highest level menu of the execution monitor. They affect the general state of the monitor.

- Help (H)**           Accesses the NMODE help system. Information on the monitor can then be obtained with the Where-am-I (**W**) command.
- Options (O)**       Gives you a choice of entering a browser for debug printing options (**P**) or execution monitor options (**M**). The execution monitor options are described below.
- Breakpoints (B)**   Enters a menu for changing the status of the monitor's breakpoints (i.e., functions the monitor will stop at when calling or returning). The following sub-commands are available.  
Edit (**E**): Enter a new list of breakpoint functions.  
Mark (**M**): Set a break at the return of the highlighted form.  
Unmark (**U**): Remove a breakpoint from the highlighted form.  
Activate (**A**): Enable stopping at breakpoints (the default).  
Deactivate (**D**): Disable stopping at breakpoints.  
Quit (**Q**): Return to the main monitor command level.
- Quit (Q)**           If the top-level form being stepped has returned, quitting turns off monitoring and selects the previous buffer. If the stepping is not complete and the user confirms the request, quitting finishes evaluation of the current form with monitoring turned off. Return values of currently active forms will be shown, but new evaluations within the form will not be monitored.
- Abort (A)**          After prompting for confirmation, abort stops monitoring of new evaluations. If there is an active form, abort terminates stepping without returning a value; this has the same effect as the command bound to Lisp-A.

## Stepping Commands

These commands control the stepping process. The stepping always proceeds from the last form in the execution monitor; it does not matter what line is currently highlighted.


- Evaluate (E)**       This command evaluates the current form without monitoring it, and displays the results.
- Step (S)**           Take the next step. One of two things happens:
- If the form is a call to a compiled function that doesn't call any interpreted functions, the function is called, the result displayed, and the next form to be stepped is displayed.
  - Otherwise, the next form to be stepped is displayed.

- 
- Walk (W)** Slowly steps automatically until you press `Return` or a breakpoint is reached, displaying the steps as it proceeds.
  - Run (R)** Steps automatically until you press `Return` or a breakpoint is reached. No intermediate forms are displayed, but they are stored in the monitor's history and can be accessed with the `>` command.

### Information Commands

The following execution monitor commands control the information that is displayed.


- More (>)** If the highlighted item is a previously executed form, show an added level of detail of the steps of its execution. In this case, the `>` command can be repeated to show more detail. If the item is a value, prettyprint it to the output buffer.
- Less (<)** Remove detail about the highlighted item from the display.
- View (V)** Show more information about the highlighted item. This is a two level command that is sensitive to the type of the highlighted item. The available sub-commands are described below.



When the highlighted line is a function call, the following view commands are available.

- Code (C)** If the highlighted item is a call to a function or method whose definition is in a loaded code browser, the definition is displayed.
- Values-and-args (V)** Displays the function's arguments and their values, one per line. If the function has returned, it also displays the value(s) it returned. This command is used if you want to select an argument or value to prettyprint, get, or modify.
- No-values (N)** If the highlighted item is a function whose arguments and value(s) are displayed on individual lines, it removes the argument and value lines from the display.
- Quit (Q)** Exits the View command menu.

When the highlighted item is a form to be stepped, the following view commands are available.

- 
- Prettyprint (P)** Prettyprints the form to the output buffer.
  - Values (V)** Displays the value(s) returned by a stepped form, one per line. This command is used if you want to select a value to prettyprint, get, or modify.



No-values (N)            If the highlighted item is a form whose value(s) are displayed on individual lines, it removes the value lines from the display.

Quit (Q)                Exits the View command menu.

If the current item is a function's argument or value, the following View commands are available.

Prettyprint (P)        The current value is prettyprinted.

Value (V)              Prompts for the name of a special variable, and then stores the value of the item into that variable.

Set (S)                Prompts for a new value for the highlighted item. Only allows changes to arguments of a function about to be called.

Quit (Q)              Exits the View command menu.

## Program Errors

If an error occurs in the monitored execution, the message will be printed in the output buffer (as usual). The monitor display will be correct if you were single stepping or walking, but not if you were running. After the error, you can do anything you could normally do in a debug listener or break loop.

If the error is continuable and you wish to continue, type **Lisp-C** or the monitor's Step, Walk, or Run commands. At this point a prompt for input may be printed in the output buffer. If so, you must move to a Lisp buffer and execute the form that you wish to respond with.

If the error is not continuable, either use the monitor's Quit command or **Lisp-Q** or **Lisp-A** to quit out of the break loop.

## Options

The execution monitor options browser lets you inspect and modify aspects of the stepping environment. Some of the options (such as "Breakpoints list") may also be modified by execution monitor commands. To change the value of the highlighted option, use the Browse/modify (B) command. You will either be prompted for a new value or the value of the option will be toggled. The available options are:

1. **What to monitor:** Lets you choose between stepping all expressions or only function calls. If `nil`, then no intermediate results are shown.
2. **Maximum monitoring depth:** When the depth of evaluation nesting goes beyond this number, the execution will continue, but will not be monitored.
3. **Show rarely used options:** Toggles whether or not to display the options described after this one.
4. **Stop at breakpoints:** Toggles whether the the monitor will stop at breakpoints.
5. **Breakpoints list:** Prompts for you to enter a new value for the breakpoints list.
6. **Maximum display indentation:** The monitor will not indent items further than this column; it will restart indentation in column two.
7. **Number of steps kept in history:** The minimum number of most recent history items always kept by the execution monitor. History for returned items older than this is periodically pruned.
8. **Number of steps before pruning:** Determines how often the monitor will remove old history.
9. **Keep histories of previous runs:** If `Keep`, then the monitor maintains histories of the previous monitoring of forms.

---

## The Execution Stack Browser

The execution stack browser is an interactive tool that lets you examine and alter the Lisp execution stack. Like the execution monitor, it appears as an NMODE browser, and is not available outside the NMODE environment. When you are in a debug listener and the stack browser is loaded, you can browse the execution stack with the command **Lisp-b (C-J b)**. The stack browser is most useful when executing a function signals an error and you want to know why. It is loaded with

```
(require "debug-br")
```

In the stack browser you can see the sequence of function calls that were active when the break occurred. There are commands to display information about the active functions, such as




- A description of the parameter list.
- The values of arguments.
- The values of local variables

This information is only available for interpreted functions. The only visible component of a compiled function is its name on the stack.

### Commands

Once you have gotten into a debug listener and browsed into the stack browser, the following commands are available:

Help (H)	Enters the standard NMODE browser help facility. Information on the stack browser can then be obtained with the <b>Where-am-I (W)</b> command.
Browse (B)	If the highlighted item is an activation of a function or method, the browser searches for the function or method definition in any code browsers that are present; if the definition is found, it is browsed into. If the highlighted item has a value (such as a lexical or special variable), then the function <code>describe</code> is called with the value.
More (>)	Displays more detail about the highlighted item. For a function, the first level of detail is the values of its parameters. Pressing > a second time displays the values of local variables.
Less (<)	Reverses the effect of the last > command on the highlighted item.
Value (V)	Prompts for the name of a special variable and then stores the value of the highlighted item into that variable.

- 
- Set (S) Prompts for a value to be stored into the highlighted item. This can be a form to be evaluated; the return value will be used.
  - Disassemble (D) Writes the disassembly of the highlighted function into the output buffer.
  - Options (O) Gives you of choice of two options browsers — one for changing the values of the debugging `*print-level*`, `*print-depth*` and `*print-radix*`, and one for determining whether certain kinds of functions show up in the stack browser.
  - Look (L) Reexamines the stack, and displays any changes. This is not usually needed, since the stack is examined upon entry to the browser, but can be used to update the display after a change in the environment, such as setting a new current package.
  - Quit (Q) Exits the execution stack browser.
- 
- 

---

## The Inspector

The term “inspector” refers to a tool for examining and altering Lisp data (i.e., objects). HP’s Lisp system provides a command-based inspector.

The Common Lisp function `inspect` is used to enter the command-oriented object inspector. This is actually a listener mode that defines a set of read macros (the “commands”) to access inspecting functions.

`(inspect object)`

*Function*

This writes information about *object* to the stream `*standard-output*`, and puts you into an alternate listener for the inspector. (See the section “Alternate Listeners” at the beginning of this chapter or the “Listeners” section of the “Concepts” chapter). You can then enter commands to obtain more information, or to alter the object being inspected. Commands are entered by evaluating forms that are preceded by the current listener macro character. **The following documentation assumes that the macro character has been set to its default (! : the exclamation point) by evaluating**

`(system:on 'listener-read-macro)`

In addition to using the read macros as commands, you can also call the inspecting functions directly.

The general form of output from the command-style inspector is:

```
#<ITEM: address> : item-type
field-name      : field-value
                :
```

A few of the inspector commands require you to specify the part of the object that the command applies to. This is done with the name of the field, possibly followed by indices into the value of the field (if it’s an array or list). See the example for an illustration of using an index in a command.

## Commands

The commands available in the inspector are described below. All of the commands can be abbreviated by using the first letter of the command name. If you specify arguments in a command, the command must be a list, otherwise the parentheses can be omitted. The name of the function that a command maps to is given at the end of the command description. Remember that this documentation assumes that the listener macro character has been set to the exclamation point.

!?

*Inspect Listener Macro*

Displays a list of the inspector commands. Takes no arguments. Expands into a call to the function `debug:inspect-?`.

!(describe [*field-name index*])

*Inspect Listener Macro*

Prints information about a field of the inspected object. Takes one optional argument: the name of the field to be described. If no arguments are specified, then the current object being inspected is described. Expands into a call to the function `debug:inspect-describe`.

!(inspect *field-name index*\*)

*Inspect Listener Macro*

Inspects the field *field-name*. If optional indices are given, then the array or list element specified by that index or indices is inspected. This differs from the `!describe` command in that the `!inspect` command recursively invokes the inspector on the specified field, not just describes it. Expands into a call to the function `debug:inspect-inspect`.

!(modify *field-name index*\* *value*)

*Inspect Listener Macro*

Changes the value of a field of an inspected object. The arguments to `modify` are the name of the field to be changed, an index or indices into the field if the inspected object is a list or an array, and the new value for the specified item. Note that *value* is evaluated. This command expands into a call to the function `debug:inspect-modify`.

!source

*Inspect Listener Macro*

If the object being inspected is a function object, either the disassembly of the function is displayed (if the function is compiled), or a form equivalent to the preprocessed definition of the function is displayed (if the function is interpreted). Expands into a call to the function `debug:inspect-source`.

**!top**

*Inspect Listener Macro*

Takes you back to inspecting the object that was the original argument to the function `inspect`. Expands into a call to the function `debug:inspect-top`.

**!up**

*Inspect Listener Macro*

Takes you up one level in an inspection. This is useful if you've used the `inspect` command to inspect a field of an object and now want to get back to that object. Expands into a call to the function `debug:inspect-up`.

**!quit**

*Common Listener Macro*

Leaves the inspector listener. Expands into a call to the function `system:listener-quit`.

**!abort**

*Common Listener Macro*

Returns you to the top-level listener. Expands into a call to the function `system:listener-abort`.

## Inspecting Instance Types

With the inspector you can quickly see what methods and instance variables are defined by an instance type. Inspect the symbol that identifies the instance type and then inspect the `Object-Type` field. You then have a choice of inspecting either `Methods` or `Instance-Vars`.

## Example

Here's a transcript of an inspection session. Forms evaluated by the user are preceded with a `>` for clarity.

```
> (system:on 'listener-read-macro)
#\!

> (inspect '(a b c))
#<ITEM: #x806F4F9C> : LIST
Length           : 3
Value            : (A B C)

> !?
?                Print this text
(describe item)  Print description of item
(inspect item)   Recursively inspect
(modify item value) Change the value of item
(source)         Display source (or assembly)
(quit)           Quit the inspector
(up)             Move up one level
(top)            Go to top level inspect
Commands without arguments do not require parentheses
The inspect and modify commands can take dimensions following item
```

```
> !(modify value 0 'd)
> !d
#<ITEM: #x806F4F9C> : LIST
Length           : 3
Value            : (D B C)

> (defstruct foo x y)
FOO

> (setq afoo (make-foo :x '(a b c) :y 8))
#S(FOO X (A B C) Y 8)

> (inspect afoo)
#<ITEM: #x40D6CA78> : STRUCTURE
Structure-Type    : FOO
X                 : (A B C)
Y                 : 8

> !(i x)
#<ITEM: #x80D6C730> : LIST
Length           : 3
Value            : (A B C)

> !u
#<ITEM: #x40D6CA78> : STRUCTURE
Structure-Type    : FOO
X                 : (A B C)
Y                 : 8

> !(m x 0)

> !d
#<ITEM: #x40D6CA78> : STRUCTURE
Structure-Type    : FOO
X                 : 0
Y                 : 8

> !q
```



---

## Debug Listener Mode

The debug listener is another debugging tool. It is entered when an error is signalled or `break` is called. It provides a set of commands (read macros) and functions to obtain information about the state of the system. Most of these deal with information that is stored on the execution stack. Note that unlike the browser-oriented debuggers, this capability can be used without the NMODE programming environment. However, to make these functions are available, you must load the `debug` module with

(require "debug")

If the debug module is not loaded when an error is signalled, you will be put into a simple break loop, which is a listener with less capabilities than the debug listener. Break loops are described later in this chapter.

### Debug Listener Commands

Some commands are atoms, some are lists. The function that a command maps to is listed along with the command. Commands that accept variable or argument names quote the names for you. If you call the function directly, you must quote the names. Remember that this documentation assumes that the listener macro character has been set to the exclamation point with (`system:on 'listener-read-macro`).

(`debug:backtrace`)  
`!b`

*Function*  
*Debug Listener Macro*

Displays a simple function level execution stack backtrace. Functions are listed starting at the bottom of the stack, so the stack is displayed with the top of the stack at the bottom of the display. An interpreted function is indicated by a lowercase "i" before the function name; otherwise, assume the function is compiled. An attempt is made to determine who called `break` or signalled the error, and this function becomes the current selected item. The current selected item is indicated by an arrow, (->) to the left of the item. The current selected item may be moved by the commands `!u` and `!d` (descriptions of these follow).

The `!b` command normally suppresses most system functions that it considers to be of no interest to the user. Finer control over stack items visibility can be achieved with the `!show` and `!hide` commands discussed below. `!b` expands into (`debug:backtrace`).

`(debug:up-fn [count] [name])`  
`!(uf [count] [name])`

*Function*  
*Debug Listener Macro*

The `!uf` command moves the pointer to the current selected item up the stack (towards the top of the execution stack) to the function item specified. Specifying `count` moves that many function items. `Name` may be either a symbol or a string. If it's a symbol, then `!uf` moves up to a function item whose name is eq to the symbol given. If `name` is a string, `!uf` moves up to a function item whose name contains the given string as a substring (case insensitive comparison). If both `name` and `count` are specified, then `!uf` finds the `count`th occurrence of a function environment satisfying `name`. When both `name` and `count` are given, they may be given in either order. If neither `count` or `name` are given, then `!uf` can be used instead of `!(uf)`.

`!(uf 2 foo)` expands into `(debug:up-fn 2 (quote foo))`.

`(debug:down-fn [count] [name])`  
`!(df [count] [name])`

*Function*  
*Debug Listener Macro*

The `!df` command works the same as `!uf`, only it searches down the stack. It expands into a call to `debug:down-fn`.

`(debug:up [count] [name])`  
`!(u [count] [name])`  
`!(~ [count] [name])`

*Function*  
*Debug Listener Macro*  
*Debug Listener Macro*

The `!u` (or `!~`) command moves the current item pointer to the stack item specified. Behavior is the same as the `!uf` command except that `!u` deals with any item on the stack display (not just function items). If `name` is given, it is compared against function names **and** lexical environment names. For debugging purposes, it can be useful to put a `(declare (name the-name))` in any closures so that they can be easily identified when they appear on the stack. If neither `count` nor `name` is given, then the command `!u` can be used instead of `!(u)`. `!(u 3 foo)` expands into `(debug:up 3 (quote foo))`.

`(debug:down [count] [name])`  
`!(d [count] [name])`

*Function*  
*Debug Listener Macro*

The `!d` command works like `!u` except that it moves down the stack instead of up. It expands into a call to the function `debug:down`.

`(debug:top)`  
`!top`

*Function*  
*Debug Listener Macro*

The `!top` command makes the top function item from the current backtrace the current selected function environment. `!top` expands into `(debug:top)`.

```
(debug:more-detail)
!>
```

*Function*  
*Debug Listener Macro*

If the selected item is an interpreted function, the !> command displays the chain of active lexical environments for that function. The display includes the lambda list (with &optional and &rest indicators, init forms and supplied-p vars) and let lists with init forms. !> expands into (debug:more-detail).

```
(debug:less-detail)
!<
```

*Function*  
*Debug Listener Macro*

The !< command eliminates the display of the active lexical environments for the current function item. !< expands into (debug:less-detail).

```
(debug:value var)
!(v var*)
!(s {var value}+)
```

*Function*  
*Debug Listener Macro*  
*Debug Listener Macro*

The !v command returns the value of the given variable in the current selected lexical environment (item). If a requested variable is not accessible from the current environment, then nil is returned and a diagnostic message is displayed. When !v is used to access the value of a special variable, the value returned is the value of that special in the debugger's environment, not its value in the environment being debugged. This behavior for special variables may change in future releases. Note that debug:value is a valid place for setf. !(v x y) expands into

```
(values (debug:value (quote x)) (debug:value (quote y)))
```

The !s command allows you to set the value of variables in the current selected lexical environment. The setting of specials with !s is not currently supported. !(s x 1 y 2) expands into

```
(setf (debug:value (quote x)) 1
      (debug:value (quote y)) 2)
```

```
(debug:arg arg)
!(arg arg+)
!(sa {arg value}+)
```

*Function*  
*Debug Listener Macro*  
*Debug Listener Macro*

The commands !arg and !sa work like !v and !s, except that they implicitly operate on the lambda environment of the current function item, instead of the currently selected lexical environment. !(arg p1 p2) expands into

```
(values (debug:arg (quote p1)) (debug:arg (quote p2)))
```

!(sa p1 11 p2 "fleas") expands into

```
(setf (debug:arg (quote p1)) 11
      (debug:arg (quote p2)) "fleas")
```

`(debug:step-from-listener { :continue | :quit })`  
`!step`  
`!qstep`

*Function*  
*Common Listener Macro*  
*Common Listener Macro*

The `!step` and `!qstep` commands exit the current listener and enable stepping in the execution monitor. You must be running NMODE to use these commands. If the execution monitor is not loaded, a continuable error will be signalled. Continuing from this error will load the monitor and then begin stepping. The commands differ in the way the current listener is exited. The `!step` command exits with a continue operation, while `!qstep` exits with a quit. `!step` expands into `(debug:step-from-listener :continue)`; `!qstep` expands into `(debug:step-from-listener :quit)`.

`(debug:show-classes {class}*)`  
`!(show {class}*)`  
`(debug:hide-classes {class}*)`  
`!(hide {class}*)`

*Function*  
*Debug Listener Macro*  
*Function*  
*Debug Listener Macro*

The `!show` and `!hide` commands control what is displayed when you are viewing the execution stack. There are four classes of functions normally considered uninteresting to the user. Classes are identified by their names, which are keywords. The classes are:

- `:primitive`      Functions which manipulate primitive data objects in the system.
- `:interpreter`    Interpreter functions — functions that would not be called if the function being interpreted was a compiled function.
- `:debugger`        Debugger functions.
- `:system`          All other functions used in the implementation of the Lisp system.

A class is made visible by specifying it in a `!show` command, and hidden by specifying it in a `!hide` command. For the first three classes, if a function is a member of the class, it is displayed if the class is visible, and not displayed if the class is hidden. If the system class is hidden, only system functions called from user code are displayed.

`!show` expands into `(debug:show-classes)`; `!hide` expands into `(debug:hide-classes)`.

---

## Tracing a Function

It is sometimes useful to see the sequence in which some particular functions are called during the execution of a program. Common Lisp provides the ability to mark a particular function as **traced**, so that when it is called, its name is displayed along with the values of its arguments. When a traced function returns, its value(s) is/are displayed. Hewlett-Packard has extended this capability to allow tracing of methods as well as functions. Tracing has also been extended to allow a traced function to do more than just display its name and the values of its arguments. Breakpoints can be conditionally set, and arbitrary forms evaluated when a traced function is called.

`(trace [trace-spec])`  
`(untrace [trace-spec])`

*Macro*  
*Macro*

The macro `trace` is used to set trace options for a function or set of functions. The macro `untrace` is used to remove tracing options from a function or set of functions.

*Trace-spec* is one or more function specifiers followed by zero or more trace options. A call to `trace` adds the specified options to the tracing behavior of the given functions, `untrace` removes them. A function specifier can be one of three things:

1. A symbol identifying the function to be affected.
2. A list (`:method instance-type-name method-name+` ) which indicates that all the named methods of the given instance type will have the trace options added or removed.
3. A list (*trace-spec*). This allows nesting of trace specifiers. The semantics of this are described shortly.

All tracing options involve the evaluation of a form. The form is evaluated in an environment with the following special variable bindings in effect:

<code>debug:*name*</code>	The symbol naming the function being traced.
<code>debug:*self*</code>	When tracing an ordinary function, <code>*self*</code> is bound to <code>nil</code> . When tracing a method, it is bound to the instance the method is being invoked upon. For ordinary functions, this should never be set to anything besides <code>nil</code> .
<code>debug:*args*</code>	A list of the arguments passed to the traced function.
<code>debug:*depth*</code>	The depth of recursive calls to the function since tracing began. (Value is 1 for first call.)

**debug:\*values\*** Useful only for options that evaluate forms after return from the traced function. It is bound to a list of the values returned by the function. For “before” forms, it is bound to (). Note that for functions that return zero values, it will also be bound to () for “after” forms.

The available tracing options are:

- :breakb** [*predicate*] The optional form *predicate* (T if omitted) is evaluated before calling the traced function. If true, the function **break** is called.
- :breaka** [*predicate*] The optional form *predicate* (T if omitted) is evaluated after returning from the traced function. If true, the function **break** is called.
- :break** [*predicate*] The equivalent of specifying both **:breakb** and **:breaka** with the same *predicate*.
- :before** *form* *Form* is evaluated immediately before calling the traced function.
- :after** *form* *Form* is evaluated after returning from the traced function.
- :both** *form* The equivalent of specifying both **:before** and **:after** with the same *form*.
- :trace-output** [*predicate*] The optional form *predicate* (T if omitted) is evaluated before and after calling the traced function. If nil, the normal, system-generated trace output is not written. This is useful in cases where break predicates or before/after forms can provide sufficient debugging information and the normal trace output would be redundant.

When no options are specified in a trace specifier argument to **trace**, normal tracing behavior is established, i.e., each time the affected functions are called, output containing information about the call and subsequent return is written to the stream that is the current value of the variable **\*trace-output\***.

If called with no parameters, **trace** returns a list of all functions currently traced along with their active options. If called with no parameters, **untrace** untraces all functions in the system.

## Nested Trace Specifiers

When trace specifiers are nested, as in

```
(trace foo (fee fie :before (print *important-var*))
           :no-trace-output :after (my-test))
```

then a given option applies to all functions in the trace specifier in which it appears, plus any functions nested within that trace specifier. So in the above example, `fee` and `fie` have both `before` and `after` forms with normal trace output turned off; `foo` has an `after` form with normal trace output turned off. In the case of conflicting options, such as

```
(trace foo (oof :before (check-that-thing)) :before (lets-take-a-look))
```

the innermost option prevails. So `oof` has the `before` form `(check-that-thing)`, and `foo` has the `before` form `lets-take-a-look`.

Specifiers at a given level are effectively evaluated left to right, so the rightmost prevails if conflicting specifiers are given.

## Changing Options

Options are cumulative. Any call on `trace` adds the specified options to the active set for a given function. Previously specified trace options which are independent of the new ones remain in effect. A new specification for an option already in effect supplants the previous specification. For example,

```
(trace x :breaka *sometimes*)
```

sets a conditional breakpoint right after the execution of `x` based on the value of `*sometimes*` at the time of exit. If followed by

```
(trace x :breaka *occasionally*)
```

then the break will be based on the value of `*occasionally*` instead of `*sometimes*`.

## Tracing Order

The evaluation of a traced function `foo` has seven steps. Depending on the trace options in effect, some of these steps may be skipped, but the order is consistent.

1. Write values of arguments to `*trace-output*`. If there is a `:trace-output` option, its predicate is evaluated and if nil, no output is written.
2. Evaluate “before” break predicate (`:breakb`) and break if true.
3. Evaluate “before” forms (`:before`).
4. Call the “real” `foo`.
5. Evaluate “after” forms (`:after`).
6. Write result value(s) to `*trace-output*`. If there is a `:trace-output` option, its predicate is evaluated and if nil, no output is written.
7. Evaluate “after” break predicate (`:breaka`) and break if true.

This ordering has several implications. Normal trace output will show the original arguments supplied to `foo` by the calling program, and the result value(s) that the calling program receives back. These are also the values that will be visible inside a debug listener entered during the tracing process. This means that the before and after forms are essentially part of the definition of `foo`. If a before form modifies the arguments, or the after form modifies the result, there is no way to distinguish this from the actual behavior of `foo`. The following two functions can be used when you want the trace output to show the actual behavior of a traced function even when modifications may be made by before and/or after forms.

`(debug:trace-entry-print)`  
`(debug:trace-return-print)`

*Function*  
*Function*

These functions get the function name, arguments, and value(s) from `debug:name*`, `debug:args*`, and `debug:values*` respectively, and write them to `*trace-output*` in a manner consistent with normal trace output. For traced methods, the value of `debug:self` is also written. By turning off normal trace output with the `:trace-output` option and calling these functions at the appropriate place in the before and after forms, the trace output will display the actual behavior of the traced function.



## Examples

Here are some examples of setting and unsetting trace options. Assume that the forms are being evaluated in order.

```
(use-package 'debug)           ; Avoid package qualifier for
                               ; *args* and *values*

(trace foo1 foo2 foo3)         ; Normal tracing of three functions.
(untrace foo1)                 ; Disable tracing of foo1.
(trace foo2 (foo3 :breakb)     ; Add an after form to foo2 and foo3,
  :after (print *args*))       ; break before foo3. Normal output.
(trace foo2 :breaka)           ; Break after foo2, retain after form.
(untrace foo3 :breakb)        ; Remove break before foo3.

(trace (:method shape          ; Normal tracing of :rotate and :shade
  :rotate :shade))           ; methods for shapes.

(untrace (:method shape :rotate)) ; Disable tracing of :rotate method.

(trace)                        ; Show all tracing options in effect.
(untrace)                      ; Remove all tracing options from everything.
```

## Miscellany

When using trace be aware of the following things:

- If a function that is currently being traced is redefined by `load`, `compile`, `defun`, or `(setf (symbol-function...)...)`, only the basic definition is modified; tracing behavior is maintained.
- Only symbols can be traced, not function objects. If `func` is a function that is currently being traced, evaluating  

```
(apply 'func (list x y z))
```

will be traced just like `(func x y z)`, but  

```
(apply #'func (list x y z))
```

will result in no tracing behavior.
- Functions called from within break predicates and before/after forms are not traced.

---

## The Break Loop

The break loop was explained earlier in this chapter in the “Concepts” section. Most of the time you are developing code you will have loaded the `debug` module, so will be using the debug listener, which is an enhanced break loop. This section explains some of the details about the break loop and related variables.

In the current system implementation, the break loop listener is equivalent to the top loop listener. No distinction is made between break loops entered by an error condition and those entered by a user call on the function `break`.

The following listener commands and NMODE commands are available in a break loop. (The listener commands assume that you have set up the default listener macro character with `(system:on 'listener-read-macro)`).

Command	NMODE command	Action
<code>!a</code>	<code>Lisp a</code>	Abort to top loop
<code>!c</code>	<code>Lisp c</code>	Continues from the current listener
<code>!q</code>	<code>Lisp q</code>	Quits from the current listener

## Break Loop Related Variables

As with many parts of the Lisp system, the behavior of the break loop depends in part on the values of some global variables. This section describes the variables pertinent to the break loop. The symbols naming these variables are all in the `sys` package.

`system:*break-hook*`                      Initial Value: `break-loop`                      Variable  
`system:*default-break-hook*`            Initial Value: `break-loop`                      Variable

The variable `*break-hook*` must always be assigned. Its value should be the name of a function of no arguments, which when called establishes the actual break loop for the user. NMODE redefines this variable. Many subtleties of the current break implementation must be understood before this variable should be redefined. The value of `*default-break-hook*` is initialized to the same value as `*break-hook*`. It is provided for the convenience of developers who wish to redefine `*break-hook*`, so they can still call the default break hook while doing some additional processing before or after calling it.

`system:*break-level*`                      Initial Value: 0                                      Variable  
`system:*break-level-limit*`                Initial Value: 10                                     Variable

The variable `*break-level*` is rebound and incremented on each new entry into a break loop. It tells how many levels deep the system currently is in break loops. After `*break-level*` is incremented, but before the break hook is called, `*break-level*` is compared with `*break-level-limit*`. If it is greater, the break hook is not called, and a diagnostic is printed noting the failed attempt to enter another level of break. This test may be defeated by setting `*break-level-limit*` to `nil`.

<code>system:*break-limit-exceeded*</code>	Initial Value: 0	Variable
<code>system:*break-limit-exceeded-limit*</code>	Initial Value: 50	Variable

Each time the break hook is called, `*break-limit-exceeded*` is rebound to 0. Each time `*break-level-limit*` is exceeded, the variable `*break-limit-exceeded*` is incremented by one. Each time `*break-limit-exceeded*` is incremented, it is compared against the value of `*break-limit-exceeded-limit*`. If less, nothing happens. If equal, a warning is issued. If greater, `sys:listener-abort` is called, leaving you in the top level loop. None of this testing and drastic behavior occurs if `*break-limit-exceeded-limit*` is `nil`.

The primary reason for setting `*break-limit-exceeded-limit*` to a non-`nil` value is for terminating non-interactive Lisp sessions which encounter error conditions that try to query a user.

If `*break-limit-exceeded-limit*` is non-`nil`, the net effect of all this is as follows. Assume `*break-level-limit*` is 5 and `*break-limit-exceeded-limit*` is 10. On entering each of the break levels 1 through 5, `*break-limit-exceeded*` is rebound to 0. Each time we attempt to enter break level 6, `*break-limit-exceeded*` is incremented, until on the tenth attempt you are warned that the next attempt will cause a listener abort. If you then cause another error which attempts to enter another break loop, `sys:listener-abort` is called. If at any time while at break level 5, we exit the break loop, going back to level 4, `*break-limit-exceeded*` is reset by the restoration of the old binding. Thus, succeeding entries to break level 5 get a fresh start with `*break-limit-exceeded*` set to zero.

These two variables also play a part in protecting you from a particular infinite looping situation. If the listener makes two consecutive calls to `read` that do not make any “progress”, it increments `*break-limit-exceeded*` and then does so every subsequent time that its call to `read` does not make “progress”. As soon as “progress” is made, `*break-limit-exceeded*` is reset to 0. If `*break-limit-exceeded*` becomes greater than `*break-limit-exceeded-limit*`, `sys:listener-abort` is called. Lack of “progress” usually means reading an end of file.

<code>*break-on-warnings*</code>	Initial Value: <code>nil</code>	Variable
----------------------------------	---------------------------------	----------

Works as documented in Steele’s *Common Lisp*. If `nil`, it affects nothing. If non-`nil`, causes all calls to the function `warn` to enter a break loop.

# File System Dependencies

---

## Introduction

Because its designers expected Common Lisp to be implemented on a wide variety of hardware and operating systems, the definition includes a system-independent mechanism for specifying files and/or directories. This facility is based on data objects called **pathnames**. This chapter describes how HP-UX file names are mapped to Lisp pathnames in Hewlett-Packard's implementation of Common Lisp.

It also describes how we have chosen to implement several functions whose behavior is not given explicitly in the Common Lisp definition because they are almost inherently system-dependent. These functions deal with the loading of Lisp files and modules: **provide**, **require**, and **load**.

---

## Pathnames

Common Lisp pathnames have six components: host, directory, name, type, version, and device. Namestrings (strings that identify a file in the manner used by HP-UX, such as `"/users/bonzo/.login"`) are parsed to produce the various pathname components. The methods for parsing a namestring to produce the pathname components read like the rules to some sort of bizarre game, but they should be made clear by the examples that follow these descriptions.

- Host** Identifies the type of file system. This is always `"HP-UX"`. The host component of a pathname can be obtained with `pathname-host`.
- Directory** Identifies the directory. This is a list of strings that are the names of the directories in the namestring. If the namestring begins with a `/`, then the first string in the list is `"`. Anything between a pair of `/`'s is taken to be a directory name, so if a namestring ends with a `/`, then the name component is `nil`. The directory component of a pathname can be obtained with `pathname-directory`.
- Name** A string representing the basename of the file. This is everything in the namestring to the right of the rightmost `/` (or the beginning of the namestring if there are no `/`'s) and to the left of the rightmost period (or the end of the namestring if there are no periods). If the rightmost period comes immediately after the rightmost `/`, then it is assumed to be part of the name rather than the delimiter between the name and type. The name component of a pathname can be obtained with `pathname-name`.
- Type** A string representing the type of a file. This is everything in the namestring to the right of the rightmost period, unless the rightmost period falls at the beginning of the name, in which case the type is `"`. The type component of a pathname can be obtained with `pathname-type`.
- Version** This component is not used. Its value defaults to `nil`. You can obtain the version component of a pathname with `pathname-version`, but it won't be very interesting.
- Device** This component is not used. Its value defaults to `nil`. You can obtain the device component of a pathname with `pathname-device`, but it won't be very interesting.

## Examples

A few examples should clear up any confusion you have about the way namestrings are parsed to produce pathname components. If you're not sure about the way a particular string would be parsed, just try it.

```
(pathname-name "bin/rm") => "rm"
(pathname-name "/jose/the/amazing/wonderdog/") => ""
(pathname-name "lispcode/source/hobnail.l") => "hobnail"
(pathname-name ".") => "."
(pathname-name "$HOME/.cshrc") => ".cshrc"

(pathname-directory "/usr/bin") => (" " "usr")
(pathname-directory "usr/bin/") => ("usr" "bin")
(pathname-directory "/users/orion/quiver/arrow.l") => (" " "users" "orion"
                                                    "quiver")

(pathname-type "bin/kazbah.b") => "b"
(pathname-type "..") => ""
(pathname-type "pompons.asc") => "asc"
```

## Resolving Filenames

Notice that these rules only specify the mapping of namestrings to pathnames. They do not say anything about how the system "finds" the files that namestrings or pathnames identify. In HP-UX, file names are either **relative** or **absolute**. An absolute file name is one that begins with a / (which means to start at the "root" directory). A relative file name is one that does not begin with a /. The search for the file corresponding to a relative file name begins at the current directory. When you're in the Lisp system, the current directory is the one that was current when you invoked the system, or possibly one that was subsequently changed to by calling the HP-UX function *chdir(2)* from Lisp.

Namestrings for files may contain HP-UX shell variable references such as `$HOME` or `$mysource`. These are properly expanded when it comes time to access the file, but are **not** expanded when the namestring is parsed for its pathname components. For instance,

```
(pathname-directory "$HOME/.nmoderc") => (" $HOME")
```

---

## Loading Modules

It is convenient to think of Lisp code in terms of **modules**, with each module being a unit of code that provides some particular service. Common Lisp provides several functions that deal with modules, but the details of how they operate is left to the particular implementation. This section describes how Hewlett-Packard has chosen to define three Common Lisp functions that deal with modules and code files.

`(load pathname &key :verbose :print :if-does-not-exist)` *Function*

The `load` function is used to load code files into the Lisp system. It can be used to load Lisp source (`.l`) or binary (`.b`) files, or HP-UX a.out files (`.o`) (`.o` files can only be loaded with `load` if the `external` module is loaded). If *pathname* is a namestring with a suffix or a pathname with a type, then the specific file identified by *pathname* is loaded in the appropriate manner. If no suffix or type is specified, the system looks for a file to load in the following way:

1. If there is a file named *filename* with no suffix, assume that it is a Lisp source file and load it.
2. If there is a file named *filename* with a `.l` suffix, and one with a `.b` suffix, load the one that has been written most recently.
3. If there is a file named *filename* with a `.l` suffix, load it.
5. If there is a file named *filename* with a `.b` suffix, load it.
6. If there is a file named *filename* with a `.o` suffix, assume that it is an a.out file and load it with `extn:load-ofile`.

Each step assumes that the conditions for the preceding steps were not satisfied.

It is not possible to load from a stream in HP's implementation of Common Lisp.

`(provide module-name)` *Function*

A call to `provide` could be placed at the beginning of the file that implements a module. When called, it adds *module-name* to the list of modules maintained in the variable `*modules*`. *Module-name* is a string or symbol. If it's a symbol, the print name of the symbol is used as the module name. If you put the `provide` at the beginning of a file, the module will be registered as loaded even if an error causes the load to be aborted. If you put the `provide` at the end of a file to avoid this problem, make sure that there is no way for the module to require itself (directly or indirectly), or you will initiate an infinite cycle of loads.

(require *module-name* &optional *pathname*)  
system:\*require-directories\*

Function  
Variable

The function `require` is used to make sure that a particular module is loaded. When `require` is called, `*modules*` is searched for *module-name*. If it's not found and the optional argument *pathname* (a pathname or list of pathnames) is specified, then the file(s) identified with the *pathname(s)* are loaded as if `load` was called with them as parameters. If the required module was not found in the loaded files (i.e. no `provide` of the module was executed), a warning is issued.

If the module name in a call to `require` is not found in `*modules*`, then the system assumes that the module is implemented in a file with the same name as the module name. It then searches the directories whose pathnames are in the list `system:*require-directories*` for the appropriate file to load. Exactly which file to load is determined in the manner described above for `load`.

System modules to be loaded with `require` are normally stored in subdirectories of the `$LISP/modules` directory. The default value of `sys:*require-directories*` is a list of pathnames corresponding to the `local`, `lisp`, `extn`, `lib`, and `nmode` subdirectories. User-defined modules you would like to be able to load with `require` should be put in the `$LISP/modules/local` directory. Then you will not have to specify the optional *pathname* argument to `require`.

## Multiple File Modules

If a module is implemented in more than one file, there are two ways to make sure that a call to `require` loads all the files that implement that module. The first way is to always provide the optional argument to `require` that names the files to be loaded. So if you have a module `dance` that is implemented in the files `dance.l`, `twist.l`, and `rhumba.l` (all of which are in the directory `$arts`), the call to `require` would be

```
(require "dance" '("$arts/rhumba.b" "$arts/twist.b" "$arts/dance.b"))
```

Remember that the order of the files to be loaded may be important. The call to `provide` for the module should be in the rightmost file in the list of files to be loaded (`$arts/dance.b` in this case).

The other way to require a module implemented in multiple files is to put calls to `load` in the file that has the same name as the module. So the beginning of `dance.l` would be

```
(load "$arts/rhumba")  
(load "$arts/twist")  
(provide "dance")
```



Then to require the dance module, you would do the following:

```
(unless (member "$arts" *require-directories*) ; This would probably be  
  (push "$arts" *require-directories*))      ; done somewhere else.
```

```
(require "dance")
```

## Introduction

Because of implementation dependencies, the Common Lisp standard does not define some functions that are useful (even necessary) for Lisp programmers. For this reason, Hewlett-Packard has added some functions to its version of Common Lisp. Some of these extensions are discussed in the appropriate chapter (such as the debugging functions), but the ones that do not fall into a convenient category are covered here.

---

### NOTE

Since the functions described in this chapter are HP extensions, you should not use them if you want your code to be strictly portable. If you must use them, but may port your code in the future, try to isolate them as much as possible.

---

---

## System Functions

The following functions manipulate the state of the Lisp system. They are all in the `system` package.

`(system:gc)`

*Function*

Calling `system:gc` initiates garbage collection and returns `nil` when garbage collection is complete. Call the function `room` to see how much space is available.

`(system:exit)`

*Function  
Variable*

`system:*exit-forms*`

Calling `system:exit` terminates the current Lisp session. Prior to termination, the forms in `system:*exit-forms*` are evaluated. It makes no difference if you're in the NMODE environment, or just the bare language interpreter; Lisp is terminated, and you are left in the shell from which Lisp was invoked. When using NMODE, we recommend that you use the `C-X Z` command instead of calling `system:exit`.

The variable `system:*exit-forms*` is a list of forms to be evaluated before the Lisp process terminates. This list should include forms to release operating system resources (windows, for example) that were allocated during the session.

`(system:save-world file &optional init-forms message)`

*Function  
Variable*

`system:*save-world-init-forms*`

A call to `system:save-world` saves the current state of the Lisp system in *file* (a string, pathname, or symbol whose print name is used) and then returns `nil`. The resultant file is called a **dump file**. *Init-forms* is a list of forms to be evaluated immediately upon restoration of the dump file. This list of forms is appended to the list of forms in `system:*save-world-init-forms*`. All of the forms in the resulting list are evaluated sequentially when the dump file is restored. *Message* is a string or symbol (print name is used) that is to be displayed when the dump file is restored. The current date is appended to the message before the system is saved. *Message* defaults to "Saved World". The `save-world` function cannot be called from within NMODE.

After calling `save-world`, you can later restore that state by executing the file from a shell. For example, if you evaluate

```
(system:save-world "/users/fido/bin/mylispenv")
```

and then leave Lisp, you can type

```
/users/fido/bin/mylispenv
```

to reenter the Lisp environment that was saved.

The variable `*save-world-init-forms*` is a list of forms to be evaluated when a saved Lisp system is restored (see above). The initial value of `*save-world-init-forms*` is implementation dependant. Forms can be added to this list, but you should not indiscriminately delete forms from it.

A good use for `*save-world-init-forms*` is to specify forms necessary for reinitializing modules when they require special actions when restoring a dump file (such as reinspecting the state of the file system, or regaining access to an operating system resource). When such a module is loaded, it should append the necessary forms to `*save-world-init-forms*`.

Creating a dump file does not save the dynamic state of your Lisp system. Special variables will have the values they had when `save-world` was executed, but outer bindings are lost. Similarly, execution in the restored system will no longer be within the dynamic scope of any functions, catches, when-errors, or unwind-protects that were active when `save-world` was called.

---

## Operating System Access Functions

The functions, macros, and variables described in this section provide access to the operating system that the Lisp process runs on. Some of these have functionality **similar** to that provided by the NMODE environment. The functions here are not meant to supplant the NMODE facilities, but to provide programmatic HP-UX access, and access when not running with NMODE.

`(system:current-directory)`

*Function*

A call to `current-directory` returns a pathname that is the name of the directory currently being used as the file system default. This directory can be changed by using `setf`. For example,

```
(setf (system:current-directory) "/tmp")
```

The current directory can be set to a string, symbol, pathname or `nil`. If `nil`, then the users home directory is made the current directory.

`(system:get-program-args)`

*Function*

A call to this function returns a simple vector of strings containing the arguments specified on the operating system command line when Lisp was invoked.

`(system:host-command-function cmd &rest args)`

*Function*

`(system:host-command cmd &rest args)`

*Macro*

`system:*host-command-symbol-format*`

*Variable*

`system:*host-command-quoted-format*`

*Variable*

`system:*host-command-object-format*`

*Variable*

The function `host-command-function` is a primitive command interpreter that invokes programs on HP-UX without terminating the Lisp process. The Lisp process is suspended while the program is executed, and then the Lisp system resumes execution. The value returned is the termination status of the program. All of the arguments to `host-command-function` are strings: `cmd` names the program or command to be executed; the rest of the arguments are the arguments for `cmd`. The value of the user's `PATH` HP-UX environment variable is used to search for `cmd`.

Since the command is not part of the Lisp process, any I/O performed by the command is independent of Lisp. This means that the values of `*standard-input*` and `*standard-output*` are irrelevant. If Lisp is using the same terminal that the command uses, their output may be interspersed. Lisp programs that maintain screen appearances (such as editors) may want to clear the screen before invoking a host command, and repaint it afterwards. Note that when NMODE is running, the output from `host-command-function` goes to the terminal or window from which NMODE was invoked, not to an NMODE buffer.

The `host-command` macro is provided to make it easier to invoke an HP-UX command or program. Its arguments have the same meanings as those for `host-command-function`, but they can be things other than strings. The arguments are converted to strings and passed to `host-command-function`.

The following rules are used to convert `host-command` arguments to strings. The left column indicates the type of the argument; the right column says what is done to convert the argument.

Strings	No conversion is performed.
Symbols	Symbols are written to a string with <code>format</code> and the format string in <code>*host-command-symbol-format*</code> .
Quoted Expressions	The quoted item is written to a string with <code>format</code> and the format string in <code>*host-command-quoted-format*</code> .
Conses	Conses whose car is something other than <code>quote</code> are assumed to be evaluable forms. These are evaluated just before the call to <code>host-command-function</code> and the result is written to a string with <code>format</code> and the format string in <code>*host-command-object-format*</code> .
Other Objects	Other objects (e.g. numbers, characters) are written to a string with <code>format</code> and the format string in <code>*host-command-object-format*</code> .

The initial values for the various format strings are as follows.

```
*host-command-symbol-format* ⇒ "~(~a~)" ; lower case, no escapes
*host-command-quoted-format* ⇒ "~a"      ; no case conversion, no escapes
*host-command-object-format* ⇒ "~a"      ; no case conversion, no escapes
```

Two significant implications of these format strings:

- The names of symbol arguments are made lowercase when converted to a string.
- A symbol returned by a form argument will not be made lowercase when converted to a string.

If the user is accustomed to typing commands to a command interpreter that provides helpful aliasing or text substitutions, he may find unexpected results using `host-command-function` and `host-command` since these will not perform the same services. However, the user may invoke the command interpreter and pass it the appropriate command string, thus getting the behavior he expects. If this is the behavior most often desired, the user is encouraged to write his own function to do this, such as

```
(defun ci (command-string)
  (system: host-command-function "csh" "-c"
    command-string))
```

## Command Examples

Here are a few examples of calls to `host-command-function` and `host-command`.

```
(host-command-function "cp" ".nmoderc" "/users/novice/.nmoderc")  
(host-command cp .nmoderc /users/novice/.nmoderc)
```

```
(setq file-to-print "foofile")  
(host-command-function "lp" file-to-print)  
(host-command lp (eval file-to-print))
```

---

## Error Signalling and Handling

Some extensions have been made to the definitions of several of the Common Lisp functions for signalling errors. This section describes these extended functions, as well as some macros that let you use the extensions.

---

### NOTE

This mechanism for handling errors is preliminary, and subject to change in subsequent releases of the system.

---

## Defining Error Symbols

`(extn:defferror symbol error-format &key continue-format)`

*Macro*

Symbol is not evaluated; all other arguments are evaluated. The `defferror` macro declares *symbol* to be a valid symbol to be used in place of the *format-string* argument to the functions `error`, `error` (actually, the *error-format-string* argument to `error`), `break` and `warn` and the macros `check-type` and `assert`. The `defferror` macro stores the string *error-format* where it may be found by the error functions. If the keyword parameter `:continue-format` is given, its value is also saved to be used by `error` as described below. The benefit of using `defferror` to associate format strings with symbols is described below under "Error Handling".

The existence of `deferror` allows the following extensions to these Common Lisp functions and macros:

<code>(error {symbol   format-string} &amp;rest args)</code>	Function
<code>(cerror {symbol   continue-format-string error-format-string} &amp;rest args)</code>	Function
<code>(warn {symbol   format-string} &amp;rest args)</code>	Function
<code>(break &amp;optional {symbol   format-string}&amp;rest args)</code>	Function
<code>(checktype place typespec &amp;optional {symbol   string})</code>	Macro
<code>(assert test-form [({place}*))</code>	Macro

In each case, if a symbol is provided instead of a string, the appropriate association made by `deferror` is used to yield a format string. The `cerror` case is slightly different. The Common Lisp specification states that `cerror` takes two string arguments and a rest argument. When called with a symbol parameter, the symbol takes the place of both strings. Both needed string values must have been declared by `deferror` (the error format string as the first parameter and the continue format string as the `:continue-format` keyword parameter). The arguments to `cerror` must either be a symbol followed by the `&rest` parameters, or two strings followed by the `&rest` parameters. The macros `check-type` and `assert` do not evaluate the symbol or string argument.

## Error Handling

The value of these error signalling extensions lies in their ability to support error identification by error handlers. The macros and functions described in this section are used for error handling. The symbols that name these functions and macros are all in the `extn` package. When compiling or interpreting functions that use the macros described here, you must load the `exception` module with

```
(require "exception")
```

At run time however, all the functions needed are in the basic Lisp system.

```
(extn:when-error form {handling-form}*)
```

 Macro

The macro `when-error` evaluates `form` under the “protection” of an error detection mechanism. If no errors occur, `when-error` returns the value(s) produced by `form`. The `handling-forms`, if any, are not executed. If, however, during the evaluation of `form`, an exception is signalled by a call to either of the functions `error` or `cerror`, `handling-forms` will be evaluated. Before evaluation of the handling forms is begun, the variable `*exception-info*` is bound to an object containing information about the exception that caused the handling forms to be executed. See the subsequent section *Exception Information* for details on what information is available and how it is accessed.



The handling forms of the `when-error` are evaluated without benefit of special error protection. If handling forms are specified, `when-error` returns whatever is returned by the last handling form. Providing no handling forms effectively says to ignore all exceptions; in this case `when-error` returns `nil` if an exception occurs.

`(extn:break-on-errors {form}*)`

Macro

The macro `break-on-errors` will override any enclosing `when-error` and enter a break loop if an error occurs while evaluating the *forms*. Continuing from a break loop entered within the dynamic scope of a `break-on-errors` behaves like a normal `continue`. Quitting from such a break loop, however, causes `break-on-errors` to return `nil` (so the handling forms of an enclosing `when-error` will not be evaluated in either case).

Both `break-on-errors` and `when-error` implicitly set up a catch that is thrown to when an error is signalled.

### Exception Information

As described in this chapter, the variable `*exception-info*` is at times bound to a value containing exception information. In order to allow for future extensions or modifications, the exact form of this value is not defined. Instead, functions are provided to allow you to access the various “fields” of `*exception-info*`, and to construct an exception information value that is compatible with the accessor functions. It is an error to rely on a specific format for the value of `*exception-info*`.

`(extn:exception-function &optional exception-info)`

Function

`(extn:exception-arguments &optional exception-info)`

Function

`(extn:exception-continuable-p &optional exception-info)`

Function

`(extn:exception-signaller &optional exception-info)`

Function

`(extn:exception-symbol &optional exception-info)`

Function

Each of these functions operates on the variable `*exception-info*` unless *exception-info* is supplied (in which case that value is used). *Exception-info* should of course only be constructed by calling the function `exception-info` described below.

The function `exception-function` returns the symbol naming the error function that was called. Currently, this will be either `error` or `error`, though there could be others in the future.

The function `exception-arguments` returns a list of all the arguments that were passed to the function named by `exception-function`.

The function `exception-continuable-p` returns `true` if the exception is continuable, and `nil` otherwise.

The function `exception-signaller` returns the symbol naming the function which signalled the exception. If the name of the function cannot be determined, then the symbol `:unknown` is returned.

If the exception was signalled using a symbol declared by `deferror`, `exception-symbol` returns that symbol. If a symbol was not used in the call by the exception signaller, `nil` is returned.

It is at this point that the symbol parameter for the error functions becomes useful. If the error handler wishes to identify or classify an error, the only information available is that which is actually passed to the error function. Parsing format strings to identify errors would be difficult, at best. Arbitrarily complex error identification/classification is now possible by associating whatever information an implementation/application desires with a symbol.

If you need to create your own exception information value, the function `exception-info` must be used.

*(extn:exception-info func arglist signaller continuable?)*

*Function*

This uses its arguments to construct a value that has the same format as `*exception-info*`. *Func* is the symbol identifying the function that was used to signal the exception (error or `error`). *Arglist* is a list of arguments that would be acceptable to *func*. *Signaller* is a symbol identifying the function that caused the exception to be signalled, and *continuable?* is a boolean indicating whether or not the exception is continuable.

### **User Defined Exception Handlers**

The system allows you to define and use your own error handling function through the use of a variable, `*exception-hook*`, which is initially `nil`. When non-`nil`, the value of `*exception-hook*` should be an error-handling function (typically a symbol that identifies a function) that takes one argument.

If `*exception-hook*` is non-`nil`, and an exception is signalled, the handler function identified by `*exception-hook*` is called with the appropriate exception information value as its argument. While the handling function is executing, `*exception-info*` is bound to the exception information value. The handler is always called, even if the error occurs within the scope of a `when-error` or `break-on-errors`. To minimize the risk of infinite recursion, `*exception-hook*` is rebound to `nil` immediately before the user defined error handler is invoked. If this is not what you desire, you must explicitly reset `*exception-hook*` within the body of your handler.

There are three legitimate ways to exit from a user defined exception handler.

`(extn:exception-decline)`

*Function*

If the handler determines that it does not wish to handle the exception, it should call `exception-decline`. In this case, the system will proceed with whatever action would have occurred if `*exception-hook*` had been `nil` when the exception was originally signalled.

`(extn:exception-continue)`

*Function*

If the exception is continuable, the handler may correct the error that caused the exception and then call `exception-continue` to continue the computation that signalled the exception. An error will be signalled if `exception-continue` is called when the exception being handled is not continuable.

`(extn:exception-quit &optional exception-info)`

*Function*

The effect of calling `exception-quit` within an exception handler is to throw to the nearest exception handling form (such as a `when-error` or `break-on-errors`). If you want your handler to throw a value other than `*exception-info*`, use `exception-info` to construct an exception value and pass that value as the optional argument to `exception-quit`. Note that the Lisp top loop executes under the “protection” of a `break-on-errors`, so that there will always be an active handler to throw to, even if you have not explicitly provided one.

An error is signalled if either of the functions `exception-decline` or `exception-continue` is called outside the dynamic extent of an exception hook function. The function `exception-quit` may be called at any time to throw to the nearest enclosing error handler.

### **Exception Messages**

`(extn:exception-msg &optional exception-info)`

*Function*

The function `exception-msg` returns a string containing the complete message that was (or would have been) printed for the current exception. If called from a break loop or user-defined exception handler, the message is the one for the exception that caused the break loop or handler to be entered. If `exception-msg` is called in the handling forms of a `when-error`, the message is for the exception being handled, even though the exception environment has been exited.

This function is useful both in an interactive mode, where it can be used to determine why a break loop was entered, and programmatically, where user-defined handlers can use it to report the appropriate condition.

The function `exception-msg` uses the current value of `*exception-info*`, unless the optional parameter `exception-info` is supplied. When supplied, it must be an exception information value constructed with the function `exception-info`.

## Error Handling Example

The following is a short transcript of a session that demonstrates these error features.

```
1 LISP [USER:] > (require "exception")
"exception"
2 LISP [USER:] > (use-package 'extn)
T
3 LISP [USER:] > (deferror my-error "I made a ~A mistake."
                  :continue-format "Forget it.")
```

MY-ERROR

```
4 LISP [USER:] > (error 'my-error 'bad)
```

```
!!!! Error: I made a BAD mistake.
      Condition signalled in INTERPRETER::LAMBDA-PUSH-FRAME
```

Entering the debugger

```
5 DEBUG (1) [USER:] > !q
6 LISP [USER:] > (cerror 'my-error "little")
```

```
!!!! Continuable error: I made a little mistake.
      Condition signalled in INTERPRETER::LAMBDA-PUSH-FRAME
```

If continued: Forget it.

Entering the debugger

```
7 DEBUG (1) [USER:] (1) > !c
NIL
8 LISP [USER:] > (when-error (error 'my-error 'bad)
                          (exception-symbol))
MY-ERROR
9 LISP [USER:] > (when-error (error "Your error.")
                          (exception-arguments))
("Your error.")
10 LISP [USER:] > (defun my-error? ()
                  (if (eq (exception-symbol) 'my-error)
                      "Shame on me."
                      "Shame on you."))
```

MY-ERROR?

```
11 LISP [USER:] > (when-error
                  (error 'my-error 'bad)
                  (my-error?))
"Shame on me."
12 LISP [USER:] > (when-error
                  (error "Your error.")
                  (my-error?))
"Shame on you."
13 LISP [USER:] > (when-error (error 'my-error 'bad))
```

NIL

```
14 LISP [USER:] > (when-error
                  (error 'my-error 'bad)
                  (format nil "~A~%But it's OK. I fixed everything."
                          (exception-msg)))
```

```
!!!! Error: I made a BAD mistake
      Condition signalled in: INTERPRETER::LAMBDA-PUSH-FRAME.
      But it's OK. I fixed everything.
```

# Notes



# Index

---

## a

!	21, 22
.	76, 77
:	76, 77
,@	76, 77
<, execution monitor command	151
<, stack browser command	154
=>	90
=>examples	93, 95, 96
!> debug listener macro	162
>, execution monitor command	151
>, stack browser command	154
!? inspect listener macro	157
!~ debug listener macro	161
'	76, 77
Abort, execution monitor command	150
!abort inspect listener macro	158
!abort listener macro	158
absolute file name	173
active	27
:after trace option	165, 167
:all-gettable option to define-type	87
:all-initable option to define-type	87
:all-settable option to define-type	87, 91
apply	40, 168
apply-method	101
!arg debug listener macro	162
arg function (:debug)	162
*args* variable (:debug)	164
argument type symbols for non-Lisp routines	127
arguments to non-Lisp routines	125, 126, 127, 128
array	48
array creation	129
array parameters to non-Lisp routines	128
array specifier for non-Lisp routine arguments	128
arrays, Fortran	142
arrays, in C	135

arrays in Pascal .....	140, 139
assembly language, example of calling from Lisp .....	143
assert macro .....	183
assignedp function (:extn) .....	110

## b

!b debug listener macro .....	160
backquote .....	76, 77
backquote examples .....	77
backtrace function (:debug) .....	160
:before trace option .....	165, 167
binding .....	8, 56
:both trace option .....	165, 167
break function .....	146, 183
Break listener mode .....	20, 23, 169, 170
break loop .....	20, 23, 169, 170
:break trace option .....	165, 167
*break-hook* variable (:system) .....	169
*break-level* variable (:system) .....	169
*break-level-limit* variable (:system) .....	169
*break-limit-exceeded* variable (:system) .....	170
*break-limit-exceeded-limit* variable (:system) .....	170
break-on-errors macro (:extn) .....	184
*break-on-warnings* variable .....	170
:breaka trace option .....	165, 167
:breakb trace option .....	165, 167
Breakpoints, execution monitor command .....	150
breakpoints:	
execution monitor .....	149, 150
trace .....	164, 165
Browse, stack browser command .....	154

## C

-c compiler option	121
C:	
arrays	135
entry point format	120
examples of calling from Lisp	133–136
pointer parameters	125, 126
call-method macro (:extn)	101, 102, 104–106
calling non-Lisp functions	119–143
catch special form	12
error function	183
character	48
check-type macro	68, 183
closure	8, 18, 19
Code, execution monitor command	151
coerce function	70
comma	76, 77
command line arguments from HP-UX	180
compilation-speed optimization quality	34
compile function	31, 38
compile-file function	38
compiled code	147
compiler	29, 31–33, 38
compiler warnings	64
concepts	7–32
consing	41–43
constant folding	34, 35
constants	42, 43
conventions	3
conversion of arguments to non-Lisp routines	125–127
:copy universal method	113, 115
:copy-instance universal method	113, 115
:copy-state universal method	113, 115
creating an instance	90
current-directory function (:system)	180



## d

!d debug listener macro	161
Debug listener mode	20, 23, 146, 160–163
debug module	160
debug-br module	148, 154
:debugger class of functions	163
debugging Lisp	145–170
declaration examples	71, 72
declarations	36, 47–72
declare special form	34, 45, 57
default methods	112–116
*default-break-hook* variable (:system)	169
*default-entry-point-format* (:extn)	123
*default-hpux-libraries* (:extn)	121
deferror macro (:extn)	182
defexternal macro (:extn)	124
defexternalvar examples	132
defexternalvar macro (:extn)	131, 132
define-entry-point function (:extn)	124
define-method examples	89, 92, 95, 96, 99–105
define-method macro (:extn)	89
define-type examples	88, 91, 95, 99, 102, 103
define-type macro (:extn)	85–88
define-type options	86–88
defining non-Lisp routines	124
defining types	53–55
defmacro macro	75
deftype macro	53–55
defvar macro	15
delayed evaluation	73
*depth* variable (:debug)	164
!describe inspect listener macro	157
:describe universal method	112
destructive functions	41
destructuring	75
device component of pathnames	172
!df debug listener macro	161
directories for loading	175
directory component of pathnames	172
Disassemble, stack browser command	155
documentation	4, 5

down function (:debug) .....	161
down-fn function (:debug) .....	161
downward closures .....	18, 38
dynamic extent .....	9-12
dynamic scope .....	9
dynamic shadowing .....	12

## e

:element-type keyword parameter to make-array .....	129
ending a session .....	178
entry point format .....	123, 137, 138
entry points .....	120-124
entry points, assembly language .....	143
entry points, Fortran .....	141
entry points, Pascal .....	138, 139
:entry-point keyword parameter to defexternal .....	124
:entry-point keyword parameter to defexternalvar .....	131
:entry-point-format keyword parameter to defexternal .....	124
:entry-point-format keyword parameter to defexternalvar .....	131
:entry-point-format keyword parameter to define-entry-point .....	124
:entry-point-format keyword parameter to load-ofile .....	122, 123, E138
entry-point-symbol function (:extn) .....	123
:eql universal method .....	112, 114
:equal universal method .....	112, 114
:equalp universal method .....	113, 114
error function .....	183
error handling example .....	187
error signalling and handling .....	182-187
errors, incomprehensible .....	29, 30, 107
eval situation for eval-when .....	31
eval-speed optimization quality .....	33-35, 38, 147
eval-when special form .....	31, 32
Evaluate, execution monitor command .....	150
examples:	
=> .....	93, 95, 96
backquote .....	76, 77
calling assembly routine from Lisp .....	143
calling C from Lisp .....	133-136
calling Fortran from Lisp .....	141, 142
calling library functions from Lisp .....	121, 136
calling Pascal from Lisp .....	137-140

closures .....	19, 38, 39
declarations .....	71, 72
defexternal .....	130, 134-143
defexternalvar .....	132
define-method .....	89, 92, 95, 96, 99-105
define-type .....	85-88
deftype .....	54
destructive functions .....	41
error handling .....	187
eval-when .....	32
free variables .....	13
inheritance .....	99, 102-106
inspect .....	159
instance initialization .....	110
load-ofile .....	122, 134-143
macros .....	78-80
object-oriented programming .....	91-96, 102-106
&optional macro parameters .....	79
pathnames .....	173
scope and extent .....	10-17
shadowing .....	11, 12
special variables .....	14-17
trace .....	168
weird errors .....	30, 31, 41-43
exception handling .....	182-187
exception handling example .....	187
exception module .....	183
exception-arguments function (:extn) .....	184
exception-continuable-p function (:extn) .....	184
exception-continue function (:extn) .....	186
exception-decline function (:extn) .....	186
exception-function function (:extn) .....	184
*exception-hook* variable .....	185
exception-info function (:extn) .....	185
*exception-info* variable (:extn) .....	183
exception-msg function (:extn) .....	186
exception-quit function (:extn) .....	186
exception-signaller function (:extn) .....	184
exception-symbol function (:extn) .....	184
execution monitor .....	148-153, 163
execution stack .....	146, 154, 160-163

execution stack browser .....	154, 155
exit function (:system) .....	178
*exit-listener-on-eof* variable (:system) .....	24
extensions to Common Lisp .....	177-187
extent .....	8-17
external module .....	119
external references .....	120-123
extn package .....	119

## f

:fast-methods option to define-type .....	87
file system .....	171-176, 180
floating point numbers .....	70
foreign functions, see “non-Lisp functions”	
Fortran:	
arrays .....	129, 142
examples of calling from Lisp .....	141, 142
parameters .....	125
free variable .....	10, 13
funcall function .....	40
function descriptions .....	3
function stepper .....	148-153, 163
function tracing .....	164-168
functional transformations .....	34-37, 39

## g

garbage collection .....	25-28, 178
gc function (:system) .....	178
gensym function .....	78-80
get-program-args function (:system) .....	180
:gettable option to instance variable specifier .....	86, 98

## h

handling errors .....	182-187
heap .....	25-28, 41-43
hide-classes function (:debug) .....	163
host component of pathnames .....	172
host-command macro (:system) .....	180
host-command-function function (:system) .....	180
*host-command-object-format* variable (:system) .....	180

<b>*host-command-quoted-format*</b> variable (:system) .....	180
<b>*host-command-symbol-format*</b> variable (:system) .....	180
HP-UX command line arguments .....	180
HP-UX, functions for accessing .....	180-182

## i

<b>ignore</b> declaration specifier .....	62
immutable data .....	42, 125
indefinite extent .....	9, 10, 13
indefinite scope .....	9-11, 13
<b>:inherit-from</b> option to <b>define-type</b> .....	88, 97-106, 116
inheritance .....	97-106
inheritance:	
<b>examples</b> .....	99, 102-106
<b>instance variables</b> .....	103
<b>methods</b> .....	98-102
<b>:methods</b> option .....	98, 104
<b>:variables</b> option .....	98, 105, 106
<b>:init</b> universal method .....	86, 108-110, 112
<b>:init-keywords</b> option to <b>define-type</b> .....	88, 98
<b>:initable</b> option to <b>instance variable specifier</b> .....	86, 108
<b>initialization keywords for instances</b> .....	90, 94, 108-111
<b>:initialize</b> universal method .....	90, 108, 109
<b>:initialize-variables</b> universal method .....	108, 109, 112
inline code .....	35, 37
<b>inline</b> declaration specifier .....	37, 62
<b>:inline-methods</b> option to <b>define-type</b> .....	87
<b>inspect</b> function .....	20, 156-159
<b>!inspect</b> <b>inspect</b> listener macro .....	157
<b>inspect-?</b> function (:debug) .....	157
<b>inspect-describe</b> function (:debug) .....	157
<b>inspect-inspect</b> function (:debug) .....	157
<b>inspect-modify</b> function (:debug) .....	157
<b>inspect-source</b> function (:debug) .....	157
<b>inspect-top</b> function (:debug) .....	158
<b>inspect-up</b> function (:debug) .....	158
<b>inspector</b> .....	156-159
<b>inspector</b> example .....	159
<b>instance type symbol</b> .....	49, 115
<b>instance variable</b> .....	84, 86, 89
<b>instance variables:</b>	

inheriting .....	98, 105, 106
pseudo .....	105, 106
instancep function (:extn) .....	115
instances .....	48, 84
instances:	
creating .....	90
initialization .....	108-111
inspecting .....	158
type .....	84
interpreter .....	33, 38, 147
:interpreter class of functions .....	163

## I

leaving the system .....	178
less-detail function (:debug) .....	162
lexical:	
scope .....	9, 10
shadowing .....	11, 12
library functions, calling from Lisp .....	121, 136
:libs keyword parameter to load-ofile .....	121
list .....	48
listener function (:system) .....	23
listener macros .....	21, 22
listener modes .....	20, 146
listener-abort function (:system) .....	24
*listener-banner* variable (:system) .....	24
listener-continue function (:system) .....	24
*listener-eval* variable (:system) .....	24
*listener-mode* variable (:system) .....	24
*listener-name* variable (:system) .....	24
*listener-print* variable (:system) .....	24
listener-quit function (:system) .....	24
*listener-read* variable (:system) .....	24
listener-read-dispatch-macro symbol .....	22
listener-read-macro symbol .....	21, 22
listeners .....	20-24
load directories .....	175
load function .....	31, 122, 174
:load-also keyword parameter to load-ofile .....	121, 122
load-ofile function (:extn) .....	121
load-related variables .....	123, 175

loading files .....	174, 175
loading non-Lisp object code .....	121
locally macro .....	57, 58
Look, stack browser command .....	155

## m

macro examples .....	78–80
macro expansion time .....	74, 107
:macro keyword parameter to <code>defexternal</code> .....	124
:macro keyword parameter to <code>defexternalvar</code> .....	131
<code>macroexpand</code> function .....	74
<code>macroexpand-1</code> function .....	74
macros .....	30–32, 73–80, 107
<code>make-array</code> function .....	42, 129
<code>make-instance</code> function .....	90
message .....	83
message sending .....	90, 91
:method breakpoint specifier .....	149
:method trace specifier .....	164
methods .....	84
:methods option to <code>inherit-from</code> .....	98, 104
methods:	
ambiguity .....	100
name conflicts .....	104
resolution .....	100
universal .....	112–116
<code>!modify inspect listener macro</code> .....	157
<code>*modules*</code> variable .....	174, 175
modules:	
<code>debug</code> .....	160
<code>debug-br</code> .....	148, 154
<code>exception</code> .....	183
explanation of .....	174–176
<code>external</code> .....	119
Monitor listener mode .....	20
monitoring execution .....	148–153
<code>more-detail</code> function ( <code>:debug</code> ) .....	162
<code>multiple-value-setf</code> macro example .....	78
mutable data .....	42

## n

name component of pathnames	172
name declaration specifier (:extn)	64
*name* variable (:debug)	164
names	8
NMODE	4, 27, 28, 148, 178, 180
:no-init-keyword-check option to define-type	88
No-values, execution monitor command	151
non-destructive functions	41
non-Lisp:	
argument type symbols	127
arguments	125-128
array parameters	128
array specifier for arguments	128
assembly language example	143
C examples	133-136
defining access functions	124
Fortran examples	141, 142
function calling	119-143
loading object files	121
Pascal examples	137-140
pointer parameters	125, 126
restrictions	130
return values	124, 130
symbol arguments	126
var parameters	125, 126
variables	131
vector specifier for arguments	128
notinline declaration specifier	62
:notinline-methods option to define-type	87

## o

.o file	120
object	82
object file format	120
object-oriented programming	19, 81-117
object-oriented programming examples	91-96
off function (:system)	22
:offset keyword parameter to defexternalvar	131
on function (:system)	21, 22
on-off? function (:system)	22



operating system access functions .....	180-182
optimization .....	33-45
optimize declaration specifier .....	34, 63, 147
&optional parameters to macros, example .....	79
options:	
-c compiler .....	121
:inherit-from to define-type .....	88, 97-106, 116
:after to trace .....	165, 167
:all-gettable to define-type .....	87
:all-initable to define-type .....	87
:all-settable to define-type .....	87
:before to trace .....	165, 167
:both to trace .....	165, 167
:break to trace .....	165, 167
:breaka to trace .....	165, 167
:breakb to trace .....	165, 167
execution monitor .....	153
:fast-methods to define-type .....	87
:gettable instance variable .....	86, 98
:init-keywords to define-type .....	88, 98
:initable instance variable .....	86, 108
:inline-methods to define-type .....	87
:methods inheritance .....	98, 104
:no-init-keyword-check to define-type .....	88
:not-inline-methods to define-type .....	87
stack browser .....	155, 163
trace .....	165, 166
:trace-output to trace .....	165, 167
:type instance variable .....	86
:variables inheritance .....	98, 105, 106

## p

parameters to non-Lisp routines .....	125, 126-128
Pascal:	
arrays .....	139, 140
entry points .....	138, 139
examples of calling from Lisp .....	137-140
procedures .....	139, 140
records .....	130, 131
var parameters .....	125, 126
pass by reference .....	125

PATH HP-UX environment variable	180
pathname examples	173
pathnames	171-173
pointer parameters to non-Lisp routines	125, 126
porting considerations	177
pp-expand function (:extn)	33, 34
predicate type specifiers	52, 53
preprocessor	29-31, 33
Prettyprint, execution monitor command	151
:primitive class of functions	163
:print universal method	112
procedures:	
Pascal	139, 140
proclaim function	34, 45, 57, 59
propagation of methods	98
*protected-entry-points* variable (:extn)	123
:protecting keyword parameter to load-ofile	121
provide function	32, 174
pseudo instance variables	105, 106

## q

!qstep debug listener macro	163
!quit inspect listener macro	158
!quit listener macro	158

## r

records, Pascal	130, 131
recovered	27
recovering from errors	24, 182-187
:redefined-methods	87, 98
redefining instance types	116
:redefining keyword parameter to load-ofile	121
relative file name	173
rename-type	116
require function	32, 175
*require-directories* variable (:system)	175
resolving file names	173, 180
restrictions on non-Lisp routines	130
:result keyword parameter to defexternal	124, 130
return value of non-Lisp routines	124, 130
Run execution monitor command	151

## S

<code>!s</code> debug listener macro	162
<code>!sa</code> debug listener macro	162
<code>safety</code>	34, 37, 130
<code>save-world</code> function (:system)	178
<code>*save-world-init-forms*</code> variable (:system)	178
saving an environment	178
Scope	8-17
<code>self</code>	89, 93
<code>*self*</code> variable (:debug)	164
<code>send?</code>	91
Set execution monitor command	152
Set stack browser command	155
<code>:settable</code>	86, 98
shadowing	11, 12
shared list structure	42
<code>show-classes</code> function (:debug)	163
<code>!source</code> inspect listener macro	157
<code>space</code>	34
<code>special</code>	11, 14-17, 47, 56, 60
special variable	11, 14-17, 56
specifying instance variables	86
<code>speed</code>	33-38
<code>stable</code>	27
<code>stack</code>	146, 154, 160-163
stack browser	154, 155
Steele	1, 47
<code>step</code>	20, 148
<code>!step</code> debug listener macro	163
Step, execution monitor command	150
<code>step-from-listener</code> function (:debug)	163
stepping functions	148-153, 163
string	48
structures:	
C	130, 131
Lisp	48
subroutines, Fortran	142
subset type specifiers	50-52
<code>substr</code> macro example	79
<code>subtypep</code> function	68
<code>supports-operation-p</code> function (:extn)	91

symbol .....	8, 13, 14, 48
symbols as arguments to non-Lisp routines .....	126
:system class of functions .....	163
system functions, Lisp .....	178-179
system-lisp declaration specifier (:extn) .....	34, 37, 39, 63

## t

the special form .....	44, 45, 65
throw special form .....	12
tools for debugging Lisp .....	145-170
!top debug listener macro .....	161
top function (:debug) .....	161
!top inspect listener macro .....	158
topics .....	2
trace macro .....	164
trace-entry-print function (:debug) .....	167
:trace-output trace option .....	165, 167
*trace-output* variable .....	165
trace-return-print function (:debug) .....	167
tracing:	
examples .....	168
functions .....	164-168
options .....	165, 166
order .....	167
type checking .....	40, 44, 45, 66-68, 130
type component of pathnames .....	172
type declaration specifier .....	61
:type option to instance variable specifier .....	86
type specifiers .....	49-55
:type-check keyword parameter to defexternal .....	124, 125, 130
type-of function .....	66, 115
typecase macro .....	69
typep function .....	66, 114
:typep universal method .....	113, 114
types .....	47-71
types of arguments to non-Lisp routines .....	127

## u

<b>!u debug listener macro</b> .....	161
<b>!uf debug listener macro</b> .....	161
<b>undefine-method</b> .....	117
<b>undefine-type</b> .....	117
<b>undefining instance types</b> .....	117
<b>*unexported-entry-points* variable (:extn)</b> .....	123
<b>universal methods</b> .....	112-116
<b>untrace macro</b> .....	164
<b>up function (:debug)</b> .....	161
<b>!up inspect listener macro</b> .....	158
<b>up-fn function (:debug)</b> .....	161
<b>upward closures</b> .....	18, 38
<b>upward-closures declaration specifier (:extn)</b> .....	34, 38, 39, 64

## v

<b>!v debug listener macro</b> .....	162
<b>Value, execution monitor command</b> .....	152
<b>value function (:debug)</b> .....	162
<b>:value keyword parameter to define-entry-point</b> .....	124
<b>Value, stack browser command</b> .....	154
<b>*value* variable (:debug)</b> .....	164
<b>Values, execution monitor command</b> .....	151
<b>Values-and-args, execution monitor command</b> .....	151
<b>:var instance variable specifier to define-type</b> .....	86
<b>var parameters to non-Lisp routines</b> .....	125, 126
<b>:var-type keyword parameter to defexternalvar</b> .....	131
<b>variables</b> .....	8
<b>:variables option to :</b>	
<b>inherit-from</b> .....	98, 105, 106
<b>variables:</b>	
<b>free</b> .....	10, 13
<b>instance</b> .....	84, 86, 89
<b>non-Lisp</b> .....	131
<b>pseudo instance</b> .....	105, 106
<b>vector specifier for non-Lisp routine arguments</b> .....	128
<b>version component of pathnames</b> .....	172
<b>version declaration specifier (:extn)</b> .....	64
<b>View, execution monitor command</b> .....	151, 152

## W

Walk, execution monitor command .....	151
warn declaration specifier .....	64
warn function .....	183
warnings, turning off .....	64
weird errors .....	29, 30, 41-43, 107
when-error macro (:extn) .....	183

# Notes



**MANUAL COMMENT CARD**

**LISP Programmer's Guide**

Manual Reorder No. 98678-90040

Name: \_\_\_\_\_

Company: \_\_\_\_\_

Address: \_\_\_\_\_

\_\_\_\_\_

Phone No: \_\_\_\_\_

Please note the latest printing date from the Printing History (page ii) of this manual and any applicable update(s); so we know which material you are commenting on \_\_\_\_\_.





NO POSTAGE  
NECESSARY  
IF MAILED  
IN THE  
UNITED STATES



**BUSINESS REPLY MAIL**  
FIRST CLASS      PERMIT NO. 37      LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Hewlett-Packard Company  
Attn: Customer Documentation  
3404 East Harmony Road  
Fort Collins, Colorado 80525







**HP Part Number**  
**98678-90040**

Microfiche No. 98678-99040  
Printed in U.S.A. 3/86



**98678-90603**  
For Internal Use Only