Part No. 070-5969-00
Product Group 07

4400 SERIES
COMMON LISP

Tektronix
COMMITTED TO EXCELLENCE

Part No. 070-5969-00
Product Group 07

4400 SERIES
COMMON LISP

Please Check at the Rear of this Manual for NOTES and CHANGE INFORMATION

First Printing APR 1986

Tektronix®
COMMITTED TO EXCELLENCE

This version describes COMMON LISP as implemented by Tektronix.

Please address all mail to:

Artificial Intelligence Machines
Tektronix, Inc.
P.O. Box 1000 M.S. 60-405
Wilsonville, Oregon 97070
Attn: AIM Documentation

MANUAL REVISION STATUS

PRODUCT: 44OO SERIES COMMON LISP PROGRAMMING

This manual supports the following versions of this product: . Version 1.0

REV DATE	DESCRIPTION
APR 1986	Original Issue

Tek COMMON LISP Installation

INTRODUCTION

This document describes the Tek COMMON LISP distribution. It describes the diskettes, the file organization, and discusses how to get COMMON LISP running on your 4400 Series system.

The software comes on a set of floppy diskettes in relative backup format. To install the software, you must create a directory called */common-lisp*, restore the files using the *restore* utility, then build your COMMON LISP system.

LOADING THE SOFTWARE

The following step-by-step procedure assumes that you have not created a file called */common-lisp* on your system.

1. Boot your system.
2. login as *system.*
3. You must be in the root directory ("/"). To insure this type:

```
chd /
```

4. Create a directory called */common-lisp*. To do so, type:

```
crdir common-lisp
```

5. Restore the COMMON LISP files from the distribution diskettes. To do so, type:

```
restore +ldb common-lisp
```

6. At the system's prompt, insert the diskettes in order.

FILE ORGANIZATION

The organization of the files are as follows:
/common-lisp:
build:
This directory contains files for building a fresh common-lisp. It contains the following:

Readme	; describes how to use the makefile.
make-cl	; builds an upper-case-insensitive COMMON LISP.
ucl	; a runtime system used only for the 4405/4406
ucl-4400	; a runtime system for either 4404 or 4405/4406.
files.bu	; contains all fasl files comprising the common-lisp system.

lib:
This directory contains two other subdirectories
code and *doc*.

/common-lisp/lib/code contains:

trace.fasl [The trace package]

step.fasl [The Stepper]

flavors.fasl [The flavors system]

vanilla.fasl [The base flavors system]

tek-graph.fasl [The 4400 Tektronix Graphics Library]

/common-lisp/lib/doc contains:

docstrings [The COMMON LISP documentation strings]

BUILDING A COMMON LISP

To build a COMMON LISP, follow this procedure:

1. restore all the diskettes as described above.
2. Move to the directory */common-lisp/build.* Type:

```
sys++ chd /common-lisp/build
sys++
```

3. Read the file called *ReadMe.* To do so type:

```
sys++ more ReadMe
```

The display will pause at the end of each screenful of text until you press the space bar.

4. Make an executable version of COMMON LISP for your machine. To make a upper-case-insensitive COMMON LISP for the 4405/4406, type:

```
sys++ make-cl
```

It will take a few minutes to create an executable file called */bin/cl.* This will be an upper-case-insensitive version (the standard version) of COMMON LISP. The file */common-lisp/build/ReadMe* contains details of how to build other case modes and versions of COMMON LISP that will run on the 4404.

5. Run COMMON LISP. To do so, type:

```
sys++ cl
```

You are now inside the Tek COMMON LISP interpreter.

Contents

Preface

The *Tek COMMON LISP User Guide* and the book, *Common Lisp: The Language*, comprise the Tek COMMON LISP documentation kit. Together, they describe the language and its use. *Common Lisp* details the functions and the calling conventions of standard COMMON LISP, while *Tek COMMON LISP User Guide* describe features of this implementation of COMMON LISP: its extensions, added features, and peculiarities. We advise the user to read at least the following chapters of this document before using Tek COMMON LISP:

Chapter 1: 'Introduction'
Chapter 2: 'Implementation'
Chapter 4: 'Operating System Interface'
Chapter 5: 'Top Level'

1 The language

The Tek COMMON LISP system consists of an interpreter and an in-line optimizing compiler. Tek COMMON LISP is a robust and complete implementation of COMMON LISP, as specified in *Common Lisp*. In addition, it has been enhanced by a fast, solid implementation of Flavors and a rich, modeless top level with extensive intrinsic debugging facilities. A symmetric interface package between LISP and extrinsic data and procedures (e.g. of C and FORTRAN) is currently under development, and will be added to the language in a later release. Tek COMMON LISP was designed to be compact and very fast. It is written in COMMON LISP and a special low-level language.

2 History

LISP was one of the first high-level computer languages developed, originating in the late fifties, soon after the emergence of FORTRAN. From the beginning LISP was a memory-intensive language. For this and less practical reasons, LISP was used mostly at universities until the parallel development of inexpensive fast memory and the nascence of microprocessors in the early eighties made it practicable for general use. By that time, LISP had diverged into a number of dialects developed at major research centers. With LISP's increasing pervasion, the LISP community felt a concomitantly increasing need

to standardize the language. This need culminated in the efforts of Guy L. Steele, Jr., and others to define a new language, COMMON LISP, combining the features of the various dialects of LISP into a single lingua franca of the artificial intelligence community, now commercial as well as academic. In this process, inconsistencies and vagaries of particular dialects were rationalized, making COMMON LISP a 'cleaner' language than older LISPs encrusted with layers of tributes to generations of hackers come and gone. The book *Common Lisp* defines the resulting language. Tek COMMON LISP is a complete implementation of the COMMON LISP language as defined in that book, enhanced with extensions in important areas such as the top level, error handling, and debugging, left undefined or vague by the book. This *Tek COMMON LISP User Guide* describes our implementation of COMMON LISP.

3 Comments and suggestions

We are always seeking a dialogue with our users in order to improve Tek COMMON LISP. We invite your comments and suggestions. A form is provided at the back of this manual for your convenience, but of course personal correspondence is always welcome. The address to which to write, either by post or by electronic mail, is on the information sheet enclosed with this document.

4 Reporting bugs

We are committed to the highest standards of software engineering. Releases of Tek COMMON LISP are extensively tested both internally and in the field before wide dissemination. Nevertheless, as with all, especially new, complicated computer programs, it is possible that you will find bugs or encounter behavior that you do not expect. In that event, we will do our utmost to resolve the problem. But, resolving bugs is a cooperative venture, and we need your help. Before reporting a bug, please study this document and *Common Lisp* to be sure that what you experienced was indeed a bug. If the documentation is not clear, this is a bug in the documentation: Tek COMMON LISP may not have done what you expected, but it may have done what it is supposed to do. A report that such and such happened is generally of limited value in determining the cause of a problem. It is very important for us to know what happed before the error occured: what you typed in, what Tek COMMON LISP typed out. A literatim log, preferably hard copy, may be needed. If you are able to localize the bug and reliably duplicate it with a minimal amount of code, it will greatly expedite repairs. It is much easier to find a bug that is generated when a single isolated function is applied than a bug that is generated somewhere when an enormous application is loaded. Although we are intimately familiar with Tek COMMON

LISP, you are familiar with your application and the context in which the bug was observed. Context is also important in determining whether the bug is really in Tek COMMON LISP or in something that it depends on, such as the operating system.

To this end, we request that your reports to us of bugs or of suspected bugs include the following information. If any of the information is missing, it is likely to delay or complicate our response.

- *Lisp implementation details.* Tell us the implementation of Tek COMMON LISP that you are using, including at least the release number and date of release of Tek COMMON LISP, the manufacturer, model, and version of the hardware on which you are running Tek COMMON LISP, and the operating system and its release number. The minimum information we need can be provided by executing the following functions within Tek COMMON LISP: **lisp-implementation-type**, **lisp-implementation-version**, **machine-type**, **machine-version**, **software-type**, **software-version**, and **short-site-name**.
- *Information about you.* Tell us who you are, where you are and how you can be reached (an electronic mail address if you are reachable via Internet or Usenet, a postal address, and your telephone number), your Tek COMMON LISP license number, and in whose name the license is held.
- *A description of the bug.* Describe clearly and concisely the behaviour that you observe.
- *Exhibits.* Provide us with the **smallest, self-contained** LISP source fragment that will duplicate the problem, and a log (e.g. produced with the **dribble** function) of a **complete** session with Tek COMMON LISP that illustrates the bug.

A convenient way of generating at least part of a bug report is to use the **dribble** function in Tek COMMON LISP. (The function is described in §25.3, p. 443 of *Common Lisp.*) First type (dribble *filename*) to record the remainder of the session in file *filename.* Then apply the functions that were described earlier to describe your implementation of Tek COMMON LISP. Next duplicate your bug. And then type (dribble) to end the log. Note that if what you type to duplicate the bug loads in files either directly or indirectly, attach a complete listing of the source version of these files to your session log. The following dialogue provides a rudimentary template for the kernel of a bug report.

```
<cl> (dribble "bug.dribble")
<cl> (lisp-implementation-type)
<cl> (lisp-implementation-version)
<cl> (machine-type)
```

```
<cl> (machine-version)
<cl> (software-type)
<cl> (software-version)
<cl> (short-site-name)
<cl> ;; Now duplicate your bug . . .
<cl> (dribble)
```

Send bug reports to either of the electronic mail or postal addresses that are given on the information sheet that is enclosed with this document. In general an electronic report can be acted upon more speedily. When we receive your bug report, it will be assigned a number by which we can mutually refer to it concisely and unambiguously, and you will be sent a receipt. We will investigate the report and inform you of its resolution in a timely manner.

We will meet you more than half way to get your project moving again when a bug stalls you. We only ask that you take a few steps in our direction.

5 Keeping abreast

We maintain mailing lists, both by post and by electronic mail. We also maintain an electronic mail forum, accessible via Internet and Usenet, for users of Tek COMMON LISP. You are invited to subscribe to our mailings and to become a member of our electronic forum. We like to hear about what our customers are doing with Tek COMMON LISP, and we can keep you abreast of new releases and other pertinent information. The addresses appear on the information sheet enclosed with this document. Join us!

1 Introduction

1 Introduction

This document is the *Tek COMMON LISP User Guide*. It is designed to supplement *Common Lisp: The Language* (Guy L. Steele, Jr., Digital Press, 1984), and it describes feature and implementation details specific to this implementation of COMMON LISP. This introduction will describe the format of this manual; the contents of the other chapters; other documents that will be useful to you, and how this document relates to them; and how to run Tek COMMON LISP and compile functions. We urge users to read this chapter, and at least chapters 2, 4 and 5 before using Tek COMMON LISP.

This document is a reference manual. Neither it nor *Common Lisp* are primers to COMMON LISP nor introductions to the language. The user is encouraged to consult textbooks on LISP such as *LISPcraft* (Robert Wilensky, W. W. Norton and Company, 1984) to gain familiarity with the language. We assume that you are familiar with at least one dialect of LISP.

1.1 Format of the manual

This document is divided up into several chapters describing how we implemented features either not described in *Common Lisp* or not specified exactly in *Common Lisp*. We have tried to follow the format of *Common Lisp*, where possible. The format is described in detail in §1.2.5 of *Common Lisp*. Briefly, definitions of functions, variables, named constants, special forms and macros appear on their own line in a special type font and in the following form:

name parameters *[Type]*

For example:

digit-char-p char ***&optional*** (radix 10) *[Function]*
default-pathname-defaults *[Variable]*
setf {place newvalue}* *[Macro]*

As in *Common Lisp*, definitions may spill over onto additional lines, and are followed by explanation and examples.

Type faces are used to distinguish between functions, symbols, constants, printed forms, and examples. Functions are printed in **bold gothic**. Other symbols are printed in gothic. Constants (such

as 0, #^A, "A", or *nil*) and special symbols (such as **package**) are printed in *italic gothic.* Keywords and lambda-list keywords (such as ***:test*** and ***&optional***, respectively) are indicated in ***bold italic gothic.*** Printed forms are printed in *italic gothic.* Examples in the text are printed in gothic. When examples appear separate from the text, output from the Tek COMMON LISP system is printed in `courier`; input to Tek COMMON LISP is printed in **`bold courier`**; and comments are printed in *`italic courier`*.

1.2 Chapter descriptions

This document contains the following chapters:

1 This 'Introduction.'

2 'Implementation.' This chapter describes specifics of this implementation of the Common Lisp language, with sections on data types, storage allocation, pathnames and the compiler.

3 'Extensions.' This chapter describes some of the extensions to Common Lisp found in Tek COMMON LISP. In particular, the error handler is described.

4 'Operating-system interface.'

5 'Top level.' The top level is the user's interface with Tek COMMON LISP. This chapter describes it.

6 'Flavors.' Flavors is an object-oriented language that is part of Tek COMMON LISP as an extension to COMMON LISP. This chapter is a user guide to the Flavors system.

7 'Profiling.' This chapter describes the profiling tools, which allow the user to discover where code is spending time.

8 'Extrinsic data and procedures.' This chapter describes the foreign-function interface.

Also included in this document are a summary of symbols and an index. Certain machine-specific information are included in the appropriate copies of this document.

1.3 Reference to other documents

Along with this document and *Common Lisp*, you should have received an installation guide. Release notes are bound in with this document. Most of Tek COMMON LISP is described in *Common Lisp*. You should refer to that book for most information on the functionality of Tek COMMON LISP. If the description in *Common Lisp* are vague, or if the implementation is explicitly unspecified, then refer to this document, where details of this implementation are described. Also found in this document are descriptions of extensions to COMMON LISP (features beyond the scope of the specification in *Common Lisp*), such as 'Flavors' (chapter 6) and 'Profiling' (chapter 7).

New users should install their system following the instructions in the installation guide. Once the system has been successfully installed, there should be no reason to refer to the installation guide again. It should be kept, however, in case you must reinstall the system for any reason.

We recommend that you at least scan most of this manual before seriously using Tek COMMON LISP, for details of the implementation are contained here. Also, look over the release notes, which list features (and problems) that affect this release. *Common Lisp* should be your main reference manual. Tek COMMON LISP implements the language as described in that book. We would appreciate any comments you have about the documentation, its completeness, ease of use, or clarity. You may send comments to the address listed on the information page at the end of this document.

1.4 How to run lisp

The installation procedure ends with the building of a file named *cl*. The exact location of this file will depend on your machine configuration. Let us assume, however, that the file is in a directory contained in your search path (e.g., */usr/local*). To get into Tek COMMON LISP, just type

```
% cl
```

There will be a short wait while Tek COMMON LISP is being loaded and initialized. Then you should see the Tek COMMON LISP header and the prompt, which looks like

```
<cl>
```

At this point, you are in the LISP environment and have all of COMMON LISP at your disposal.

If you have insufficient swap space, LISP may stop with an error message indicating that the swap space is too small. In that case, either stop other processes or increase the size of your swap space, and then start Tek COMMON LISP again. (If you find it necessary to increase your swap space, refer to your operating-system documentation.)

1.5 How to compile functions

COMMON LISP provides two ways to compile functions. The first is to define the function in the interpretive environment and then call the **compile** function. The second way is to write the function to a file either with an editor or some other means, and then call the **compile-file** function. For example, suppose you have the function **foo** defined already in your LISP environment, then to compile it, just type

```
<cl> (compile 'foo)
```

which will replace the interpreted version of **foo** with the compiled version of **foo**. If you want to compile a whole file (say *foofncs.cl*) full of functions, you can use the function **compile-file** as follows

```
<cl> (compile-file "foofncs.cl")
```

which will result in a new file being created in *foofncs.cl*'s directory called *foofncs.fasl*. This file can then be **load**ed into your LISP environment (with either the **load** function or with the **:ld** top-level command), and you will then have all of *foofncs*'s compiled functions at your disposal.

1.6 Note on special function forms

Users trying to debug code will often have occasion to look at the stack for recent function calls. There, one may find, instead of *, /, +, <, etc., oddly named functions of the form:

***string*_2op**
***string*_3op**

where ***string*** is *, +, /, <, etc. This function is called for compiler efficiency, and should be interpreted as the function named by ***string***. Thus, for example, **/_2op** should be intepreted / (i.e. the division function).

2 Implementation

2 Implementation

2.1 Data types

Tek COMMON LISP contains all of the required COMMON LISP data types plus an *instance* data type for use by the *flavors* system. *Fixnums* are signed 29-bit quantities. *Bignums* may be as large as $2^{1,048,576}$. There are two distinct floating-point types. *Short-float* and *single-float* are equivalent and are 32-bits wide. *Double-float* and *long-float* are equivalent and are 64-bits wide. The distinct array data types are the following:

```
(array t)
(array bit)
(array (unsigned-byte 8))
(array (unsigned-byte 16))
(array (unsigned-byte 32))
(array string-char)
(array single-float)
(array double-float)
(array (signed-byte 8))
(array (signed-byte 16))
(array (signed-byte 32))
(array fixnum)
(simple-array t (*))
(simple-array bit (*))
(simple-array (unsigned-byte 8) (*))
(simple-array (unsigned-byte 16) (*))
(simple-array (unsigned-byte 32) (*))
(simple-array single-float (*))
(simple-array double-float (*))
(simple-array (signed-byte 8) (*))
(simple-array (signed-byte 16) (*))
(simple-array (signed-byte 32) (*))
(simple-array fixnum (*))
```

2.2 Storage allocation

Tek COMMON LISP uses a two-space, copying garbage collector. Data objects which cannot move or are unlikely to become garbage are placed in a special static storage area.

gcprint *[Variable]*

■ If the variable **gcprint** is not *nil*, then when garbage collection is started a message will be printed on the terminal, and after it is complete another message will be printed describing the current state of storage allocation. It prints the statistics in this form: *a/b(c)*, where is *a* is the number of bytes of dynamic space is use, *b* is the total number of dynamic bytes available, and *c* is the total number of static space bytes allocated.

gc *[Function]*

■ This will cause a garbage collection to occur. A garbage collection will occur automatically whenever the free space is exhausted.

2.3 Pathnames

COMMON LISP pathnames do not always map easily into operating-system filenames. In this section we describe the mapping chosen for Tek COMMON LISP on the Unix operating system.

2.3.1 Parsing pathnames

The *host* and *version* components of pathnames are ignored.

The *directory*, *name*, and *type* fields are determined from a namestring as follows: If there are no slashes in the namestring, then the *directory* component is *nil*. If there are slashes then all characters from the beginning of the namestring to, but not including, the last slash in the namestring is the *directory* component. The one exception is that if there is just one slash and it is at the beginning of the name, then the directory component is *"/"*. After removing the directory component and the following slash from the namestring, the rest of the string determines the *name* and *type* components. If the rest of the string is empty then both components are *nil*, otherwise, the *name* contains everything up and excluding the last period in the string. The characters following the period are the *type*. If the name ends in a period, then the *type* is the empty string. The exceptions are for names beginning with a period. In this case the period is being used to hide the file from the directory listing program, not to separate the *name* and *type* components, thus a leading period is treated as a non-period character. The string *".."* is treated specially and is parsed as a *name* of *".."* and a *type* of *nil*. The following table has some examples of pathname parsing:

	Pathname components		
Namestring	*Directory*	*Name*	*Type*
"/"	*"/"*	*nil*	*nil*
"/foo"	*"/"*	*"foo"*	*nil*
"/foo."	*"/"*	*"foo"*	*""*
"/foo.b"	*"/"*	*"foo"*	*"b"*
"/foo.bar."	*"/"*	*"foo.bar"*	*""*
"/foo.bar.baz"	*"/"*	*"foo.bar"*	*"baz"*
"/foo..bar"	*"/"*	*"foo."*	*"bar"*
"foo.bar"	*nil*	*"foo"*	*"bar"*
"foo/"	*"foo"*	*nil*	*nil*
"foo/bar"	*"foo"*	*"bar"*	*nil*
"foo/bar/baz"	*"foo/bar"*	*"baz"*	*nil*
"foo/bar/"	*"foo/bar"*	*nil*	*nil*
".lisprc"	*nil*	*".lisprc"*	*nil*
"x.lisprc"	*nil*	*"x"*	*"lisprc"*
"."	*nil*	*"."*	*nil*
".."	*nil*	*".."*	*nil*
"..."	*nil*	*".."*	*""*

Table 2.1. *Examples of conversions of namestrings to pathnames.*

2.3.2 Merging pathnames

Merging of pathnames is handled specially by Tek COMMON LISP on the Unix operating system to take advantage of directory hierarchies.

Given two pathnames *a* and *b*, then the result (*c*) of merging these pathnames may cause merging of their directory components.

```
(setf c (merge-pathnames a b))
```

If pathname *a* does not have a directory component, then the directory component of pathname *b* becomes the directory component of the result *c*. If pathname *a*'s directory component is absolute (i.e. it begins with a slash *"/"*) then pathname *c* will have pathname *A*'s directory component.

If pathname *a* has a directory component that is relative, then the directory component of pathname *c* depends on the directory component of pathname *b*. If pathname *b* has a relative directory component, then *c*'s directory component will be the same as *a*'s. If *b*'s directory component is absolute, the relative directory component of pathname *a* is appended to the absolute component and the result is canonicalized to eliminate such components as *"."* and *".."*. For

example if pathname *b*'s directory component is *"/foo"* and pathname *a*'s directory component is *"./bar"*, then pathname *c*'s directory component will be *"/foo/bar"*. Finally, if pathname *b* does not have a directory component, the directory component of pathname *a* becomes *c*'s directory component.

3 Extensions

3 Extensions

3.1 Reader case modes

In Standard COMMON LISP, the reader converts all unescaped lower case characters to upper case, so that for example *Foo* and *foo* are both read as *FOO*. This is a sign of the age of the LISP programming language. When LISP was invented in the sixties, most terminals could only handle upper case. When lower case terminals started to appear, most operating systems and languages were modified to simply map lower case characters to upper case and then proceed as before. Some modern programming languages (C, Smalltalk, Modula-2, Newspeak) distinguish between upper and lower case in identifiers and programmers use this distinction to great advantage. In C, constants are usually in upper case and variables in lower case. In Smalltalk, capitalization is used to distinguish words in multi-word identifiers, and to classify identifiers. The **set-case-mode** function can change Tek COMMON LISP's reader so that the case of characters in identifiers is significant.

There are two parameters that determine the reader's actions: *case preference* and *case sensitivity*. The preferred case is either *upper* or *lower*, and refers to the case of the characters in the print names of all of the standard symbols, such as **car** and **cdr**. Case sensitivity is either *sensitive* or *insensitive*. Case-sensitive means that the reader doesn't modify the case of any characters it reads. Case-insensitive means that characters that are not of the preferred case are converted to the preferred case.

3.1.1 The modes

There are four possible values for the combination of case preference and case sensitivity.

- *Case-insensitive, upper case preferred.* This is the mode used in Standard COMMON LISP and in most of the older LISPs such as MacLISP. With this mode you can even enter LISP programs with a card punch.
- *Case-sensitive, upper case preferred.* This is the mode used by InterLISP. This is perhaps the most difficult mode to use since you have to hold the shift key down to type the names of system functions, or else they won't be recognized.

- *Case-insensitive, lower case preferred.* This mode is very similar to the case-insensitive, upper case preferred mode.
- *Case-sensitive, lower case preferred.* This is the mode use by FRANZ LISP (and the C programming language). It matches the conventions of the Unix and Tektronix operating systems, and thus is the most natural mode to use for some programmers.

3.1.2 Functions and variables

The function and variables used in switching modes are listed next.

set-case-mode new-mode *[Function]*

■ new-mode is one of the four keywords: ***:case-insensitive-upper***, ***:case-insensitive-lower***, ***:case-sensitive-upper***, and ***:case-sensitive-lower***. **set-case-mode** converts LISP to use the new mode for subsequent reading and returns a keyword denoting the previous mode. This function must do quite a bit of consistency checking when changing between modes with different case preferences, and may may take as long as three cpu minutes to complete. Below we go into more detail on the operation of **set-case-mode** and the implications on compatibility.

current-case-mode *[Variable]*

■ The value of this variable is the keyword denoting the current mode. This variable should be considered read-only, it is changed by **set-case-mode** to reflect the current mode. Its initial value is ***:case-insensitive-upper***.

ignore-package-name-case *[Variable]*

■ If this value is true, then the case of characters in single-case package names and nicknames is ignored by the reader when looking up qualified symbols. This variable is initially *nil*, but the user may find it useful to give this variable a non-*nil* value if he chooses to operate in one of the case-sensitive modes. This is described in more detail below.

3.1.3 The **set-case-mode** function

Initially the reader is in ***:case-insensitive-upper*** mode. If the user executes (set-case-mode :case-sensitive-upper), the **set-case-mode** function need only inform the reader not to alter the case of characters it reads, and inform the printer that lower case characters needn't be escaped on output. If the user wants to change the mode to one of the lower case preferred modes, then much more work must be done: Every (interned) symbol's printname is examined. If the print name

does not contain characters of different cases, then the print name is converted to the new preferred case (in this example, lower case). If the print name contains characters of different case then it isn't modified at all. If converting a symbol's print name to lower case would cause there to be two symbols with the same print name in the same package, then no conversion is done for the symbol. Similarly, the names and nicknames of packages are converted to the new preferred case if the names do not contain both lower and upper case characters. After **set-case-mode** has examined and converted as many symbol's as possible, it prints a list of those symbols which couldn't be converted due to mixed case or a symbol conflict.

3.1.4 Compatability

Changing the case of identifiers or making LISP case sensitive is not an upward compatible change to LISP. Thus the user must weigh the advantages of a more 'modern' LISP syntax against possible future drawbacks, such as not being able to run the code in other versions of COMMON LISP. Let us examine the possible compatibily problems:

3.1.4.1 Case preference

None of the standard Tek COMMON LISP code depends on the case of the characters in identifiers, and it is unlikely that future code will. Packages are unfortunately referred to by strings and thus compatibility problems can crop up. **set-case-mode** will convert the cases of package names to the preferred case. However, if lower case is the preferred case and the user types (in-package "LISP") then this will create a new package *LISP* distinct from the existing *lisp* package. There are two solutions for this problem. One is to set the **ignore-package-name-case** variable to *t*. In this case, when **in-package** looks for and doesn't find a package named *LISP*, it converts it to *lisp* and then finds the package. If **in-package** is given a name containing upper and lower case characters, then even if **ignore-package-name-case** is *t*, it will not convert it to the preferred case. For example, (in-package "Lisp") will fail to match the package named *lisp*. The second solution to this problem is to always use symbols when referring to packages, i.e. use (in-package 'lisp). This expression will refer to the *lisp* package in all modes except ***:case-sensitive-upper.***

3.1.4.2 Compiled code

When a file is compiled, the case mode setting at the beginning of the compilation is stored in the *fasl* file. If the preferred case when a file was compiled is different than than when it is loaded, the fast loader will do case conversion on the fly to those symbols whose print names which do not have both lower and upper case characters.

3.1.4.3 Using **set-case-mode**

The version of Tek COMMON LISP that you received will work in any mode. We expect that most users will do one of the following. One group will choose to use Tek COMMON LISP in its standard ***:case-insensitive-upper*** mode. These users can simply use LISP as distributed and ignore everthing about case modes. The second group will want to run in ***:case-sensitive-lower*** mode. They should execute the **set-case-mode** function, create an image with **dumplisp** and use that LISP. In order to load or compile source code written assuming the standard ***:case-insensitive-upper*** mode, they should use **set-case-mode** to put lisp in ***:case-insensitive-lower*** mode. It is much faster to go from ***:case-sensitive-lower*** mode to ***:case-insensitive-lower*** mode, than to ***:case-insensitive-upper*** mode, and the two insensitive modes are nearly equivalent in their effect.

3.2 Errors

errorset form [announcep] *[Macro]*

☐ The expression form is evaluated, and if no errors occured then the first value returned from **errorset** will be *t*, and the rest will be the values returned from the evaluation of form. If an error occurs, then the single value *nil* is returned. If announcep is non-*nil*, then the error message associated with the error will be printed, otherwise nothing is printed.

Note: This is currently how errors are detected during the evaluation of expressions. When the COMMON LISP standard specifies an error-handling system, this function may be obsoleted.

3.3 Miscellaneous functions

dumplisp ***&key :name :restart-function :read-init-file*** *[Function]*

■ Save an image of the currently executing COMMON LISP as an executable file. The name of the executable file will be *savedcl* unless the name argument is provided. Unless the read-init-file argument is given a value *nil*, when the saved image is executed, it will will search for and load the *.clinit.cl* file (see Chapter 5, 'Top level,' for a description of this file). Normally, the next step is for the top level to print a prompt and enter the *read-eval-print* loop. If however the restart-function argument is given a non-*nil* value, then that value will be **funcall**ed. Should the **restart-function** return, the standard read-eval-print loop will be entered.

uncompile function-name *[Function]*

■ If the function function-name was compiled with the **compile** function (as opposed to having been in a file that was compiled with **compile-file** and subsequently **load**ed), then the function is 'uncompiled,' i.e. its function definition is replaced by the original interpreted *lambda* form.

file-older-p file-1 file-2 *[Function]*

■ If file-1 and file-2 both exist, and if file-1 is older than file-2, this function returns *t*. Otherwise, it returns *nil*.

compile-file-if-needed filename ***&key :output-file :force-compile*** *[Function]*

■ The file filename will be compiled if it is younger than the output file specified by the ***:output-file*** keyword argument, or if the value of the ***:force-compile*** keyword argument is *t*.

□ If ***:output-file*** is not given, the compiled pathname will be constructed by merging the extension *.fasl* with filename.

pp name *[Macro]*

■ The definition of the function or macro name is 'pretty printed' to **standard-output**.

4 Operating-system interface

4 Operating-system interface

This chapter describes functions in Tek COMMON LISP that interact with the host operating system.

shell ***&optional*** command *[Function]*

■ If the command argument is not given, then an interactive shell is spawned. To get back to LISP, exit from the shell. If the command (a string) is given, then a shell is spawned and directed to execute that command.

current-directory *[Function]*

■ Return a pathname with the directory component holding the current directory.

username-to-home-directory name *[Function]*

■ Return a pathname with the directory component holding the named user's home directory.

5 Top level

5 Top level

5.1 Introduction

The user interacts with COMMON LISP in the *top level*. The top level facilitates the user's interactions with LISP, allowing the user access the full power of the LISP environment. In the top level, the user loads, debugs, and runs programs. The essence of the top level is a *read-eval-print* loop, which reads user input, evaluates it, and prints the result. Also in the top level are a set of commands which allow the user to do useful things, such as re-evaluating a previously typed command, loading files, recovering from errors and debugging.

The debugging tools in Tek COMMON LISP are integrated into the top level. They consist of a trace and step package, a set of top-level commands which allow dynamic examination and manipulation of LISP data and the runtime evaluation stack, and mechanisms to single stepping through expressions or function calls.

The Tek COMMON LISP top level is *modeless*–all top-level commands are always accessible, no matter what mode of operation the top level is currently executing (tracing a function, or stepping through expressions). For example, while stepping through the expressions in a function, the user may want to examine a functions parameters, abort stepping, ask for help on any top-level command, or exit COMMON LISP. There is one set of top-level commands which control all the functions of the top level.

5.2 The top level specification

This section describes the top-level syntax and semantics.

5.2.1 Initialization

When COMMON LISP is first invoked it looks for and loads one or more initialization files. COMMON LISP first searches the user's home directory for a file called *.clinit.cl* loading it if it is there; then COMMON LISP searches the current directory for a file of the same name, and if found, loads it too. The initialization file in the current directory overrides defaults set in the users home directory initialization file. Any valid LISP expressions may be present in the

initialization file, and it customizes the users LISP environment, by, for example, loading programs or changing reader syntax.

Top-level commands (prefixed by the top-level command character) can not be used from within the initialization file, or any other file. They may only be typed to the top level.

5.2.2 Top level input

Before reading input from the user, the top level issues a prompt. This prompt is initially the four character string *"<cl> "*. The user can change this prompt by changing the value of the variable *top-level:*prompt**. If the prompt string contains the two character subsequence *"~d"*, the top level substitutes these characters with the number assigned to the current input. See the documentation for **format** for more information on *"~d"*, and the example in §5.2.3.1 for top-level input.

5.2.3 Commands

The top level understands two sorts of input: top-level commands and LISP expressions. A top-level command is identified to COMMON LISP by prefixing it with a single character (initially the colon character). This can be changed by binding a different character object to the variable *top-level:*command-char**.

A newline typed to the top level is the null command, which is ignored, extra spaces and tabs are ignored, and typing an end-of-file has a special meaning which is discussed below (see §5.2.3.3 and §5.2.6). In Tek COMMON LISP there is no top-level command which puts the user in *debugging mode*. Debugging commands are always available–the standard function calling sequence allows for the maximum debugging information on the runtime stack.

Some top-level commands may be abbreviated–refer to the **:help** command for a list of the commands and valid abbreviations.

5.2.3.1 Getting help

:help [command-name] *[Command]*

■ Without an argument, print a summary of the commands, mostly consisting of name, abbreviation, and valid arguments; if command-name is present, then print detailed documentation about this command.

5.2.3.2 History

As the user types commands and expressions to the top level, they are recorded on an entity called the *history list*. The value of *top-level:*history** is the number of user inputs (commands or expressions) to remember, and is the maximum size the history list can

grow. When the history list reaches its maximum size, the oldest entries are thrown off as new ones are added. Note that only expressions and commands typed to the top level are added to the history list, but not input read from programs which are called from the top level.

The following commands print and retrieve expressions from the history list:

:history [***:reverse***] [n] *[Command]*

■ Print the last n, defaulting to *15*, items on the history list, in reverse order if :reverse is present.

:[+|-]number [?] *[Command]*
::pattern [? | +] *[Command]*

■ These two forms are the how previously typed expressions are re-evaluated. The first form, :number, re-evaluates the numberth typed expression, as reported by the history command. The second form, with an optional pattern, searches the history list for input matching pattern and re-evaluates the matching expression as if it were typed to the top level, otherwise, the last expression typed is re-evaluated. If + is given as an argument to the :: command form, then the search will be in the reverse sense, from the beginning of the history list forward, instead of from the end backward. If ? is an argument to either type of command, then the user will be asked to confirm the re-evaluation of the command or expression. For example:

```
<cl> (setq top-level:*prompt* "<cl ~d> ")

"<cl ~d> "
<cl 3> :his

1         (dribble "foo")
2         (setq top-level:*prompt* "<cl ~d> ")
3         :his
<cl 4> (setq a 10)

10
<cl 5> (set 'b 'setq)

setq
<cl 6> ::setq
(set 'b 'setq)

setq
<cl 7> ::(setq
```

```
(setq a 10)

10
<cl 8> :6
(set 'b 'setq)

setq
<cl 9> :his

1         (dribble "foo")
2         (setq top-level:*prompt* "<cl ~d> ")
3         :his
4         (setq a 10)
5         (set 'b 'setq)
6         (set 'b 'setq)
7         (setq a 10)
8         (set 'b 'setq)
9         :his
<cl 10>
```

5.2.3.3 Break levels

The main, or topmost, read-eval-print loop is labeled *break level 0*, and this is the level the user first enters. Each time an error occurs, a new read-eval-print loop, and thus a new break level, is entered. There are a fixed number of ways a new break level may be entered: (1) the functions **error**, **cerror**, and **break**, (2) the trace and step packages, (3) external signals, such as a keyboard interrupt, or (4) errors while reading, evaluating, or printing user input. Cases (2) through (4) are really special cases of (1)–all entrances to new read-eval-print loops are through the functions **error**, **cerror** and **break**.

When a new break level is entered a message is printed, indicating the cause, and when break levels are exited, a reminder of the previous cause is printed.

Here are the commands to manipulate break levels:

:reset *[Command]*

■ This will reset the state of the top level, and return the user to break level 0. If errors have occurred, then all error conditions will be cleared, and a **throw** to the topmost level will be done.

:continue *[Command]*

■ If the current break level is continuable, then continue computation with side effects specified by the **:error** command.

:pop [n] *[Command]*

■ Pop up to the previous break level, or to the n th previous one, if n is given.

:prt *[Command]*

■ Return to the previous break level, and retry the command which caused the error. The previous user input is printed before re-evaluation as a reminder.

:error *[Command]*

■ Print the cause of entering the current break level.

```
<cl> (setq foo bad)
Error: Attempt to take the value of the unbound sym-
bol bad.
[1] <cl> (/ 1 0)
Error: An attempt was made to divide by zero.
[2] <cl> :pop
Previous error: Attempt to take the value of the
unbound symbol bad.
[1] <cl> (setq bad :not-so-bad)

:not-so-bad
[1] <cl> :prt

<cl> (setq foo bad)

:not-so-bad
<cl> foo

:not-so-bad
<cl> (cerror "just continue" "error!")

Continuable Error: error!
If continued with :continue, just continue
[1c] <cl> :cont

nil
```

5.2.3.4 Stack commands

The *runtime stack* is the entity where arguments to LISP functions are stored. When a function is called, the calling function evaluates and pushes the arguments to the called function onto the stack. The called function then references the stack when accessing its arguments. A *stack frame* is the area on the stack where the arguments to one function call reside. If **foo** calls **bar**, which in turns calls **yaf**, then three stack frames are *active* when **yaf** is entered. The frame for

the most recently called function is on the *top* of the stack. The following commands which access and display the stack, operate on a single stack frame. After a frame is examined, it normally becomes the *current stack frame*, so further reference to the stack will, by default, operate on the previously selected stack frame. When a break level is entered, the current frame pointer starts at the top of the stack.

:zoom {arguments}* *[Command]*

■ This command prints the evaluation stack. It uses the current stack frame as the center of attention, and prints some number of frames on either side of the current frame. The value of the variable *top-level:*zoom-display** is the total number of frames to display, and an equal number of frames are printed above and below the current stack frame, if possible. The arguments to the **:zoom** command control the type and quantity of the displayed stack. After a **:zoom**, the special variable * contains the LISP expression representing the current frame. The arguments to **:zoom** are:

Only one of the following three options may be specified: (The following output control options *stick*, meaning once you use one, the next **:zoom** will use the same output style.)

:brief *[Keyword]*

☐ Print the function names of stack frames only.

:moderate *[Keyword]*

☐ Print function names, and actual parameters (the values passed on the stack). The output of this form is LISP like in appearance.

:verbose *[Keyword]*

☐ Print function names, formal (the names of the parameters in the function definition) and actual parameters.

One of the following two options may be specified:

:top *[Keyword]*

☐ Move the current stack frame pointer to the top of the stack before printing. Newer stack frames are toward the top of the stack.

:bottom *[Keyword]*

☐ Move the current stack frame pointer to the bottom of the stack before printing. Older stack frames are toward the bottom of the stack.

:n *[Keyword]*

☐ Print this many frames, instead of using the value of *top-level:*zoom-display**, which is initially 8.

:up [n] *[Command]*
:dn [n] *[Command]*

■ Move up or down the stack by n frames, or 1 if no argument is supplied. The special variable *top-level:*auto-zoom**, which defaults to *t*, controls whether a **:zoom** is done after moving the stack pointer.

:find func {options}* *[Command]*

■ Find the frame where function func is being called. The default direction to search the stack is down, or towards older stack frames. The current frame pointer is set to point to the matching stack frame, and the LISP expression corresponding to the match is bound to the variable *. The options to **:find** are:

:up *[Keyword]*
:dn *[Keyword]*

☐ Find func going up (toward newer stack frames) or down (toward older stack frames) the stack.

:skip n *[Keyword]*

☐ Skip n matching occurrences of func before setting the current frame pointer.

:current *[Command]*

■ Print the current stack frame, as a LISP expression.

:local name *[Command]*

■ This prints the value of the local (or lexical) variable name. When a variable is bound in a function by using **let**, for example, the scope of this variable is visible only inside this function. For this reason, the **:local** command is needed to examine the environments of functions on the stack.

```
<cl> (defun func (x) (car x))

func
<cl> (func 10)
Error: Attempt to take the car of 10 which is not a
cons.
[1] <cl> :zo
Evaluation stack:

->(lisp::read-eval-print-loop nil ...)
  (error)
  (car 10)
  (block func ...)
  (funcall (lambda # ...) ...)
  (eval (func 10))
  (lisp::read-eval-print-loop nil ...)
  (start-reborn-lisp)

[1] <cl> :find block
Evaluation stack:

  (lisp::read-eval-print-loop nil ...)
  (error)
  (car 10)
->(block func ...)
  (funcall (lambda # ...) ...)
  (eval (func 10))
  (lisp::read-eval-print-loop nil ...)
  (start-reborn-lisp)

[1] <cl> *

(block func (car x))
[1] <cl> :local x
10
[1] <cl> :error
current error: Attempt to take the car of 10 which is
not a cons.
[1] <cl> :current
(block func (car x))
[1] <cl> :pop
<cl> (compile 'func)

#<Function func @ #x1148c1>
<cl> (func 10)
Error: Attempt to take the car of 10 which is not a
cons.
;; notice that there is less information on the stack
;; when the function 'func' is compiled...
[1] <cl> :zo
Evaluation stack:

->(lisp::read-eval-print-loop nil ...)
```

```
    (error)
    (func 10)
    (eval (func 10))
    (lisp::read-eval-print-loop nil ...)
    (start-reborn-lisp)

  [1] <cl>
```

5.2.3.5 Miscellaneous commands

:aliases *[Command]*

■ See §5.3 for a description of this command.

:cf {file}* *[Command]*

■ The one or more arguments are interpreted as file names, which should represent the names of Tek COMMON LISP source files. The list of source files are compiled, resulting in files with the same name, but with the file type of *"fasl"*. For example, :cf foo would compile *foo.cl* into *foo.fasl*, which is acceptable to the function **load**. The files are compiled in the order they appear in the argument list. For convenience, as the previous example illustrates, the file names may be given without the file type (see chapter 23 of *Common Lisp: The Language*), which default to *"cl"*. See the compiler chapter in the Tek COMMON LISP user's manual for information on compiling files and functions, and chapter 23 of *Common Lisp* for information on *pathnames*.

☐ See also the description of *top-level:*file-ignore-case** in §5.2.6.

☐ If no arguments are given to **:cf**, then the arguments to the last call to **:cf** are used again.

:exit [val] *[Command]*

■ Exit LISP and return exit status val to the operating system or shell.

:ld {file}* *[Command]*

■ The arguments to **:ld** are interpreted as file names, which are loaded into LISP by the **load** function. If the file type portion of any of the filenames is not given and the value of the variable *top-level:*ld-options** is not *nil*, then the behavior of the **:ld** command is dependant on the value of *top-level:*ld-options**.

☐ See also the description of *top-level:*ld-options** and *top-level:*file-ignore-case** in §5.2.6.

☐ With no arguments, the last files given to the **:ld** command are loaded again.

5.2.4 The trace package

The trace package provides a way to track or trace when functions are called. For example, when tracing a function, a message is printed upon entering and exiting the function.

The trace package is invoked at the top level using **:trace** and turned off using **:untrace**. The trace package can also be invoked and exited using the functions **trace** and **untrace**, which have the same argument syntax as their top-level command counterparts (see the example below).

The output from **trace** is designed to be readable–a function being traced may be called many times, and the entrance and exit from each instance should be obvious, by the numbers at the beginning of the lines and the indentation of the lines printed by the traced function.

:trace {function-or-option-list}* *[Command]*

■ With no arguments, all the functions currently being traced are printed, otherwise the arguments to **:trace** are function names (symbols) or option lists. An option list starts with a function name, and the rest of the list are options for tracing that particular function, and do not affect the tracing of any other function. The options come in pairs, the first element of the pair being the option name (i.e., a keyword), and the second part being the option value. Missing options default to *nil*.

☐ The following are valid options to **:trace**:

:condition expr *[Keyword]*

■ Trace this function if expr evaluates to non-*nil*.

:break-before val *[Keyword]*
:break-after val *[Keyword]*
:break-all val *[Keyword]*

■ If val is *t*, then enter a new break level before, after, or before and after this function is entered. Otherwise, *val* is evaluated, in the appropriate place.

:inside func *[Keyword]*

■ Trace this function if we are currently *inside* the evaluation of the function func. func may also be a list of functions. For example, (trace (deeper :inside deep)) would trace the

function **deeper** only when called from within a call to **deep**.

:print-before expr *[Keyword]*
:print-after expr *[Keyword]*
:print-all expr *[Keyword]*

■ expr should either be a single object or a list of object which are evaluated, and the results printed before entering or after leaving the function, or both, in the case of ***:print-all***.

:untrace [function-list] *[Command]*

■ With no arguments, stop tracing all functions currently being traced, otherwise the arguments are assumed to be the names of currently traced functions which are to be untraced. **:untrace** also has a function counterpart, called **untrace**.

5.2.4.1 Special variables used by trace

The following are special variables manipulated by the trace package.

trace-output *[Variable]*

■ The stream where **:trace** sends output, which is normally **terminal-io**.

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

■ During the printing of trace forms, **print-level** and **print-length** are bound to these, respectively. See page 372 of *Common Lisp* for an explanation of **print-level** and **print-length**.

```
<cl> (defun fact (n)
(cond ((= n 1) 1)
(t (* n (fact (1- n))))))

fact
<cl> (fact 5)

120
<cl> :tra fact
fact
<cl> (fact 5)

 0: (fact 5)
   1: (fact 4)
     2: (fact 3)
       3: (fact 2)
         4: (fact 1)
```

```
          4: returned 1
        3: returned 2
      2: returned 6
    1: returned 24
 0: returned 120
120
<cl> (defun deep (x) (deeper (list x)))

deep
<cl> (defun deeper (x) (format t "~&~s~%" x))

deeper
<cl> (deep 10)

(10)

nil
<cl> :tr (deeper :inside deep)
deeper
<cl> (deeper 10)

10

nil
<cl> (deep 10)

 0: (deeper (10))
(10)

 0: returned nil
nil
<cl> :tr (deeper :break-before t)
deeper
<cl> (deep 10)

 0: (deeper (10))
Break: trace entry
[1c] <cl> :zo
Evaluation stack:

->(lisp::read-eval-print-loop t ...)
  (break "trace entry")
  (lisp::trace-call (lambda # ...) ...)
  (let (# #) ...)
  (funcall (lambda # ...) ...)
  (block deep ...)
  (funcall (lambda # ...) ...)
  (eval (deep 10))

[1c] <cl> :cont

(10)
```

```
 0: returned nil
nil
<cl>
```

5.2.5 The step package

The step package allows the user to watch and control the evaluation of LISP expressions, either inside certain functions or over certain expressions. When stepping is turned on, evaluation of all expressions is done in single-step mode–after evaluating one form, a step read-eval-print loop is entered, from which the user may continue or abort.

As with the **:trace** command, **:step** is a top-level command and **step** is a function.

With no arguments or an argument of *nil*, **step** turns off stepping. With an argument of *t*, stepping is turned on globally, otherwise the arguments are checked to be functions, and stepping is done only when inside one of the functions given to **step**.

Once stepping is turned on, the top level recognizes three more commands: **:scont**, **:sover**, and carriage return (which is a synonym for :scont 1). Also, the top-level prompt for step read-eval-print loops is prefixed with *[step]*, as a reminder that the above step commands are available.

:step [t | nil | function-list] *[Command]*

■ With no arguments or an argument of *nil*, stepping is disabled, with an argument of *t*, stepping is enabled globally, otherwise the arguments are assumed to be a list of functions wherein stepping should occur. Any non-functions supplied to **:step** will be flagged as invalid arguments, and an error will not occur.

:scont [n] *[Command]*

■ Continue stepping, for n expressions, and evaluate the last expression printed by the stepper. After evaluating the last printed expression, the next expression to be evaluated is printed. If there are no more expressions stepping is turned off. When stepping is enabled, a carriage return (or new-line) character is equivalent to :scont 1, allowing the user to step quickly with minimum keystrokes.

:sover *[Command]*

■ Evaluate the current expression in normal, non-stepping mode.

5.2.5.1 Special variables used by step

The following special variables control output from the stepper:

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

■ During the printing of forms to be evaluated, **print-level** and **print-length** are bound to the value of these variables, respectively. See page 372 of *Common Lisp* for an explanation of **print-level** and **print-length**.

5.2.6 Top-level variables

The following variables are maintained or used by the top level.

top-level:*command-char* *[Variable]*

■ The character recognized as the prefix for top-level commands. The value of this variable must be a character object, and is initially the LISP character object '#\:'.

top-level:*history* *[Variable]*

■ The number of commands which are remembered by the history mechanism, defaulting to 15.

top-level:*prompt* *[Variable]*

■ The value of this variable is printed by the top-level as a prompt for user input; it must be a LISP string. For break levels greater than 0, this prompt will be augmented with the break level number. In continuable break levels a '*c*' will be present next to the break level indicator.

top-level:*exit-on-eof* *[Variable]*

■ If bound to a non-*nil* value and the current break level is 0, then typing an end-of-file to the top-level will exit LISP, without asking for confirmation. The method of exit is taken via the function **exit**. The default value is *nil*.

top-level:*ld-options* *[Variable]*

■ The **:ld** command uses the value of this variable to determine the name of the files to load. It is intended to make the manipulation of files in Tek COMMON LISP more convenient.
□ The default value is *nil*; in this case, arguments to **:ld** are assumed to be the real file names, and given the file *foo*, **:ld** will try to load the file *foo*.

☐ In the following description of valid options, it is assumed that no file type was given–if a file type is given, the following options have no effect. For example, the input :ld foo, *foo* has an unqualified file type, whereas *foo.cl* or *foo.fasl* has a specified file type, and :ld foo.cl would load the file *foo.cl*.

:cl *[Keyword]*

☐ Look for a file with the same name but with a file type of *cl*. If this file does not exist, then try the unqualified name, with no file type. Look, for example, for *foo.cl*, and then *foo*, but not *foo.fasl*.

:fasl *[Keyword]*

☐ Look for a file with the same name but with a file type of *fasl*. If this file does not exist, then try the unqualified name, with no file type. Look, for example, for *foo.fasl*, and then *foo*, but not *foo.cl*.

:most-recent *[Keyword]*

☐ Look for the most recent (last modified) of the file types *cl* and *fasl*. For example, :ld foo would cause *foo.fasl* to be loaded if it was older, more recently modified, than *foo.cl*.

:ask-most-recent *[Keyword]*

☐ Look for the most recent (last modified) of the file types *cl* and *fasl*, and ask the user for verification before loading.

:compile *[Keyword]*

☐ This causes a compiled version of the Tek COMMON LISP source file to be loaded, compiling the source file if it is newer than the corresponding *fasl* file, or if the *fasl* file does not exist. For example, :ld foo would cause *foo.cl* to be compiled and the resulting *foo.fasl* to be loaded if *foo.fasl* did not exist or was older than *foo.cl*. In other words, this option causes an up-to-date compiled file to be loaded, compiling the source file if necessary.

:ask-compile *[Keyword]*

☐ This has the same behavior as ***:compile***, except that confirmation is obtained from the user before compiling the source file.

top-level:*file-ignore-case* *[Variable]*

■ The **:ld** and **:cf** commands use value of this variable to determine the correct file names of their arguments where the case might be incorrect because of the setting of the current case mode. (See the description of **set-case-mode** in *Tek COMMON LISP User Guide*.) *top-level:*file-ignore-case** is initially *nil*, and should either be bound to *nil* or a symbol with a **funcall**able definition, such as **string-downcase** For example, if the value of the variable **current-case-mode** is ***:case-insensitive-upper*** and **file-ignore-case** is **string-downcase**, then :ld foo/bar would look for the file *"foo/bar"*, and not *"FOO/BAR"*.

top-level:*read* *[Variable]*
top-level:*eval* *[Variable]*
top-level:*print* *[Variable]*

■ The values of these variables, if bound to valid functions (acceptable to the function **funcall**), will be **funcall**ed to read user input, evaluate the result the *top-level:*read**, and print the result of *top-level:*eval**, respectively. Great care should be taken before setting one of these variables, since binding these to something other than a function will result in a recursive error (since after an error, another read-eval-print loop is called).

top-level:*print-level* *[Variable]*
top-level:*print-length* *[Variable]*

■ *lisp:*print-level** and *lisp:*print-length** are bound to these, respectively, during the application of the *top-level:*print** function on the result of the *top-level:*eval** function. See page 372 of *Common Lisp* for an explanation of *lisp:*print-level** and *lisp:*print-length**.

top-level:*reset-hook* *[Variable]*

■ If non-*nil*, and bound to a valid function (something acceptable to **funcall**), then this function is called after executing the **:reset** command.

top-level:*zoom-display* *[Variable]*

■ The value of this variable is the number of stack frames displayed by the **:zoom** command.

Note: The following variables are required by *Common Lisp*.

+ *[Variable]*
++ *[Variable]*
+++ *[Variable]*

■ While an expression or form is being evaluated by *top-level:*eval**, the variable + is bound to the previous form read by *top-level:*read**. The variable ++ holds the previous value of +, or the form read two reads ago, and +++ holds the previous value of ++.

– *[Variable]*

■ While a form is being evaluated by *top-level:*eval**, the variable – is bound to the form itself, or the value which will be given to + after *top-level:*eval** returns.

* *[Variable]*
** *[Variable]*
*** *[Variable]*

■ While a form is being evaluated by *top-level:*eval**, the variable * is bound to the last value returned from *top-level:*eval**, or the value produced by evaluating the form in +. If more than one value is returned, all but the first are discarded, and if zero values were returned, then * is bound to *nil*. The variable ** holds the previous value of *, or the result of the second previous *top-level:*eval**, and *** holds the previous value of **.

□ If the evaluation of + produces an error, then *, **, and *** are left untouched; they are updated before *top-level:*print** is called.

/ *[Variable]*
// *[Variable]*
/// *[Variable]*

■ While a form is being evaluated by *top-level:*eval**, the variable / is bound to the list of results from the last *top-level:*eval**, or the list of all values produced by evaluating the form in +. The value of * should be the same as the car of the value of /. The variable // holds the previous value of /, or the list of results from second previous *top-level:*eval**, and /// holds the previous value of //. Therefore, the car of // should be the same as **, and the car of /// the same as ***.

□ If the evaluation of + produces an error, then /, //, and /// are left untouched; they are updated before *top-level:*print** is called.

5.3 Adding new top-level commands

The top-level command set is extensible–the user may add new commands to the list of known commands. This allows the user to further customize the top-level environment.

A top-level *alias* is a user defined top-level command, which is invoked the same as built-in top-level commands. The difference between built-in commands and aliases, is aliases can be removed, one at a time or all at once.

top-level:alias {name | (name abbr-index)} arglist body *[Macro]*

■ This is how top-level aliases are defined. The form of **top-level:alias** is similar to **defun**, and body can be anything acceptable as the body of a lambda expression. name is the name of the top-level alias, which must be a string. If the (name abbr-index) form is used, then abbr-index must be an integer which is the index into the string name which defines the shortest possible abbreviation. For example, ("load" 1) would specify that *lo* and *loa* are valid abbreviations for *load*. arglist is the list of formal parameters to the alias function, and has the same form as the formal list to a lambda expression.

top-level:remove-alias ***&rest*** names *[Function]*

■ This will remove the alias commands known by names, or all user defined aliases if the argument is ***:all***. Built-in top-level commands may not be removed with this function. If abbreviations were specified for names, then all abbreviations are also removed from the command set.

:aliases *[Command]*

■ This command prints all user defined alias commands in tabular format, with the documentation string, if there is one.

top-level:do-command name ***&rest*** arguments *[Function]*

■ This function allows the execution of top-level commands from programs. It hides the method of dispatch for top-level commands, and should be the sole means of accessing top-level commands outside typing them to the top-level read-eval-print loop.

□ name must be a string and the name of a top-level command, otherwise an error occurs.

The following function is used internally to define aliases, and is documented to allow definition of commands which can not be removed with **top-level:remove-alias**.

top-level:add-new-command name abbr handler doc *[Function]*

■ Add the string name to the list of known top-level commands. The new command will be invoked as other top-level command, by prefixing it with the top-level command character. abbr is the character index which defines the shortest abbreviation. For example, if name and abbr are *"load"* and *1*, then the **:load** command could be abbreviated to **:lo** and **:loa**. handler is the name of the function which will be called to do the work of this command, and it must be a symbol. doc is a short documentation string, which the **:help** command will print if given no arguments. Extended documentation can be specified when defining the function handler, which will be printed by the **:help** command.

```
<cl> (top-level:alias "ff" (&rest args)
"my alias for the :find command"
(apply #'top-level:do-command "find" args))

<cl> (defun test (x) (break "testing..."))

test
<cl> (test nil)

Break: testing...
[1c] <cl> :zo
Evaluation stack:

->(lisp::read-eval-print-loop t ...)
  (break "testing...")
  (block test ...)
  (funcall (lambda # ...) ...)
  (eval (test nil))
  (lisp::read-eval-print-loop nil ...)
  (start-reborn-lisp)

[1c] <cl> :ff block
Evaluation stack:

  (lisp::read-eval-print-loop t ...)
  (break "testing...")
->(block test ...)
  (funcall (lambda # ...) ...)
  (eval (test nil))
  (lisp::read-eval-print-loop nil ...)
  (start-reborn-lisp)
```

5.4 A Sample Init File

```
;;
;; common lisp initialization file
;;

(format *terminal-io* "~&; Loading home init file.")

;; the following is for the :ld and :cf commands, and so
;; that all my filenames don't have to be in uppercase!

(if (eq *current-case-mode* :case-insensitive-upper)
    (setq top-level:*file-ignore-case* 'string-
downcase))

(setq
 top-level:*prompt* "<cl ~d> "
 top-level:*history* 50
 top-level:*print-level* 20
 top-level:*print-length* 20
 top-level:*zoom-print-level* 3
 top-level:*zoom-print-length* 3
 top-level:*zoom-display* 7
 top-level:*ld-options* :ask-compile
 top-level:*exit-on-eof* t
 top-level:*command-char* #?
 top-level:*auto-zoom* nil
)

;; exit Lisp when a ^X (control-X) is typed to the top
level
(defun exit-char-mac (stream char) (exit 0))
;; the ^X in the next expression is the single character
control-X
(set-macro-character #X #'exit-char-mac)

(top-level:alias ("shell" 1) (&rest args)
  "`:sh args' will execute the shell command in `args'"
  (let ((cmd
       (apply #'concatenate 'simple-string
            (mapcar #'(lambda (x)
                        (concatenate 'simple-string
                           (write-to-string x) " "))
                    args))))
    (prin1 (shell cmd))))
```

6 Flavors

6 Flavors

6.1 Introduction

The object-oriented programming style used in the *Smalltalk* and *Actor* families of languages is available in Tek COMMON LISP. Its purpose is to perform *generic operations* on objects. Part of its implementation is simply a convention in procedure-calling style; part is a powerful language feature, called Flavors, for defining abstract objects. This chapter explains the principles of object-oriented programming and message passing, and the use of Flavors in implementing these in Tek COMMON LISP. It assumes no prior knowledge of any other languages.

The implementation of Flavors distributed by Franz Incorporated with Tek COMMON LISP is new, proprietary code which employs special interpreter and compiler hooks for very efficient execution. The code shares some components with the Franz Inc. native implementation of Flavors distributed with FRANZ LISP. Except where the underlying LISP dialects require fundamental differences (for example, in variable scoping) the two Flavors systems are functionally identical. The Tek COMMON LISP implementation of Flavors is also quite similar to that in ZetaLISP,[1] although a few details and extensions differ. Most code should port easily between the two.

The text of this chapter is a heavily-edited version of Chapter 20 from the MIT LISP Machine Manual, as made available through MIT's Project Athena. It has been subsequently edited by the staff of Franz Inc. for inclusion in the Tek COMMON LISP manual.

6.2 Objects

When writing a program, it is often convenient to model what the program does in terms of *objects*, conceptual entities that can be likened to real-world things. Choosing what objects to provide in a program is very important to the proper organization of the program. In an object-oriented design, specifying what objects exist is the first task in designing the system. In a text editor, the objects might be *pieces of text*, *pointers into text*, and *display windows*. In an electrical design system, the objects might be *resistors*, *capacitors*,

[1] ZetaLISP is a trademark of Symbolics, Inc.

transistors, *wires*, and *display windows*. After specifying what objects there are, the next task of the design is to figure out what operations can be performed on each object. In the text editor example, operations on *pieces of text* might include inserting text and deleting text; operations on *pointers into text* might include moving forward and backward; and operations on *display windows* might include redisplaying the window and changing which *piece of text* the window is associated with.

In this model, we think of the program as being built around a set of objects, each of which has a set of operations that can be performed on it. More rigorously, the program defines several *types* of object (the editor above has three types), and it can create many *instances* of each type (that is, there can be many pieces of text, many pointers into text, and many windows). The program defines a set of types of object and, for each type, a set of operations that can be performed on any object of the type.

The new type abstractions may exist only in the programmer's mind. The mapping into a concrete representation may be done without the aid of any programming features. For example, it is possible to think of an atom's property list as an implementation of an abstract data type on which certain operations are defined, implemented in terms of the LISP **get** function. There are other property lists (association lists of pairs) which are, however, not stored in the global structure of an atom, such as are implemented in terms of the COMMON LISP **getf** function. Such a property list is just a list with an even number of items. This type can be instantiated with any function that creates a list; for example, the form (list 'a 23) creates a new property list with a single key/value pair. The fact that property lists are really implemented as lists, indistinguishable from any other lists, does not invalidate this point of view. However, such conceptual data types cannot be distinguished automatically by the system; one cannot ask: *is this object a disembodied property list, as opposed to an ordinary list?*

Use of **defstruct** is another mechanism for creating new data types. This is reviewed in the next section, where a data type for ship is used as an example. **defstruct** automatically defines some operations on the objects: the operations to access its elements. We could define other functions that did useful computation with ships, such as computing their speed, angle of travel, momentum, or velocity, stopping them, moving them elsewhere, and so on.

In both cases, we represent our conceptual object by one LISP object. The LISP object we use for the representation has *structure* and refers to other LISP objects. In the case of a property list, the LISP object is a list of pairs; in the ship case, the LISP object is an array or vector whose details are taken care of by **defstruct**. In both

cases, we can say that the object keeps track of an *internal state*, which can be *examined* and *altered* by the operations available for that type of object. **getf** examines the state of a property list, and **setf** of **getf** alters it; **ship-x-position** examines the state of a ship, and (setf (ship-x-position ship) 5.0) alters it.

This is the essence of object-oriented programming. A conceptual object is modeled by a single LISP object, which bundles up some state information. For every type of object, there is a set of operations that can be performed to examine or alter the state of the object.

6.3 Modularity

An important benefit of the object-oriented style is that it lends itself to a particularly simple and lucid kind of modularity. If you have modular programming constructs and techniques available, they help and encourage you to write programs that are easy to read and understand, and so are more reliable and maintainable. Object-oriented programming lets a programmer implement a useful facility that presents the caller with a set of external interfaces, without requiring the caller to understand how the internal details of the implementation work. In other words, a program that calls this facility can treat the facility as a black box; the calling program has an implicit contract with the facility guaranteeing the external interfaces, and that is all it knows.

For example, a program that uses disembodied property lists never needs to know that the property list is being maintained as a list of alternating indicators and values; the program simply performs the operations, passing them inputs and getting back outputs. The program depends only on the external definition of these operations: it knows that if it stores a property by doing a **setf** of a **getf**, and doesn't **remf** it (or **setf** over it), then it can use **getf** to be sure of getting back the same thing which was put in. This hiding of the details of the implementation means that someone reading a program that uses disembodied property lists need not concern himself with how they are implemented; he need only understand what abstract operations are represented. This lets the programmer concentrate his energies on building a higher-level program rather than understanding the implementation of the support programs. This hiding of implementation means that the representation of property lists could be changed and the higher-level program would continue to work. For example, instead of a list of alternating elements, the property list could be implemented as an association list or a hash table. Nothing in the calling program would change at all.

The same is true of the ship example. The caller is presented with a collection of operations, such as **ship-x-position**, **ship-y-**

position, **ship-speed**, and **ship-direction**; it simply calls these and looks at their answers, without caring how they did what they did. In our example above, **ship-x-position** and **ship-y-position** would be accessor functions, defined automatically by **defstruct**, while **ship-speed** and **ship-direction** would be functions defined by the implementor of the ship type. The code might look like this:

```
(defstruct ship
  x-position
  y-position
  x-velocity
  y-velocity
  mass)

(defun ship-speed (ship)
  (sqrt (+ (expt (ship-x-velocity ship) 2)
           (expt (ship-y-velocity ship) 2))))

(defun ship-direction (ship)
  (atan (ship-y-velocity ship)
        (ship-x-velocity ship)))
```

The caller need not know that the first two functions were structure accessors and that the second two were written by hand and perform arithmetic. Those facts would not be considered part of the black-box characteristics of the implementation of the ship type. The ship type does not guarantee which functions will be implemented in which ways; such aspects are not part of the contract between ship and its callers. In fact, ship could have been written this way instead:

```
(defstruct ship
  x-position
  y-position
  speed
  direction
  mass)

(defun ship-x-velocity (ship)
  (* (ship-speed ship) (cos (ship-direction
ship))))

(defun ship-y-velocity (ship)
  (* (ship-speed ship) (sin (ship-direction
ship))))
```

In this second implementation of the ship type, we have decided to store the velocity in polar coordinates instead of rectangular coordinates. This is purely an implementation decision. The caller has no idea which of the two ways the implementation uses; he just

performs the operations on the object by calling the appropriate functions.

We have now created our own types of objects, whose implementations are hidden from the programs that use them. Such types are usually referred to as *abstract types*. The object-oriented style of programming can be used to create abstract types by hiding the implementation of the operations and simply documenting what the operations are defined to do.

Some more terminology: the quantities being held by the elements of the ship structure are referred to as *instance variables*. Each instance of a type has the same operations defined on it; what distinguishes one instance from another (besides **eq**ness) is the values that reside in its instance variables. The example above illustrates that a caller of operations does not know what the instance variables are; our two ways of writing the ship operations have different instance variables, but from the outside they have exactly the same operations.

One might ask: *but what if the caller evaluates* (svref ship 2) *and notices that he gets back the x-velocity rather than the speed? Then he can tell which of the two implementations were used.* This is true; if the caller were to do that, he could tell. However, when a facility is implemented in the object-oriented style, only certain functions are documented and advertised, the functions that are considered to be operations on the type of object. The contract from ship to its callers only speaks about what happens if the caller calls these functions. The contract makes no guarantees at all about what would happen if the caller were to start poking around on his own using **svref**. A caller who does so *is in error*. He is depending on the concrete implementation of the abstraction: something that is not specified in the contract. No guarantees were ever made about the results of such action, and so anything may happen; indeed, if ship were reimplemented, the code that does the **svref** might have a different effect entirely and probably stop working. This example shows why the concept of a contract between a callee and a caller is important: the contract specifies the interface between the two modules.

Unlike some other languages that provide abstract types, Tek COMMON LISP makes no attempt to have the language automatically forbid constructs that circumvent the contract. This is intentional. One reason for this is that LISP is an interactive system, and so it is important to be able to examine and alter internal state interactively (usually from a debugger). Furthermore, there is no strong distinction between the *system* and the *user* portions of the Tek COMMON LISP system; users are allowed to get into nearly any part of the language system and change what they want to change.

In summary: by defining a set of operations and making only a specific set of external entry-points available to the caller, the programmer can create his own abstract types. These types can be useful facilities for other programs and programmers. Since the implementation of the type is hidden from the callers, modularity is maintained and the implementation can be changed easily.

We have hidden the implementation of an abstract type by making its operations into functions which the user may call. The importance of the concept is not that they are functions–in LISP everything is done with functions. The important point is that we have defined a new conceptual operation and given it a name, rather than requiring each user who wants to do the operation to write it out step-by-step. Thus we say (ship-x-velocity s) rather than (aref s 2).

Often a few abstract operation functions are simple enough that it is desirable to compile special code for them rather than really calling the function. (Compiling special code like this is often called *open-coding*.) The compiler is directed to do this through use of macros for example. **defstruct** arranges for this kind of special compilation for the functions that get the instance variables of a structure.

When we use this optimization, the implementation of the abstract type is only hidden in a certain sense. It does not appear in the LISP code written by the user, but does appear in the compiled code. The reason is that there may be some compiled functions that use the macros (or other concrete manifestation of the implementation). Even if you change the definition of the macro, the existing compiled code will continue to use the old definition. Thus, if the implementation of a module is changed, programs that use it may need to be recompiled. This sacrifice of compatibility between interpreted and compiled code is usually quite acceptable for the sake of efficiency in debugged code.

In the Tek COMMON LISP implementation of Flavors that is discussed below, there is never any such incorporation of nonmodular knowledge into a program by either the interpreter or the compiler, except when the ***:ordered-instance-variables*** feature is used (described below). If you don't use the ***:ordered-instance-variables*** feature, you don't have to worry about incompatibilities.

6.4 Generic operations

Consider the rest of the program that uses the ship abstraction. It may want to deal with other objects that are like ships in that they are movable objects with mass, but unlike ships in other ways. A more advanced model of a ship might include the concept of the ship's engine power, the number of passengers on board, and its name. An object representing a meteor probably would not have any of these, but might have another attribute such as how much iron is in it.

However, all kinds of movable objects have positions, velocities, and masses, and the system will contain some programs that deal with these quantities in a uniform way, regardless of what kind of object is being modeled. For example, a piece of the system that calculates every object's orbit in space need not worry about the other, more peripheral attributes of various types of objects; it works the same way for all objects. Unfortunately, a program that tries to calculate the orbit of a ship needs to know the ship's attributes, and must therefore call **ship-x-position** and **ship-y-velocity** and so on. The problem is that these functions won't work for meteors. There would have to be a second program to calculate orbits for meteors that would be exactly the same, except that where the first one calls **ship-x-position**, the second one would call **meteor-x-position**, and so on. This would be very bad; a great deal of code would have to exist in multiple copies, all of it would have to be maintained in parallel, and it would take up space for no good reason.

What is needed is an operation that can be performed on objects of several different types. For each type, it should do the thing appropriate for that type. Such operations are called *generic* operations. The classic example of generic operations is the arithmetic functions in many programming languages, including Tek COMMON LISP. The + function accepts integers, floats or bignums and performs an appropriate kind of addition based on the data types of the objects being manipulated. In MACSYMA, a large algebraic manipulation system implemented in LISP, the + operation works for matrices, polynomials, rational functions, and arbitrary algebraic expression trees. In our example, we need a generic **x-position** operation that can be performed on either ships, meteors, or any other kind of mobile object represented in the system. This way, we can write a single program to calculate orbits. When it wants to know the x position of the object it is dealing with, it simply invokes the generic **x-position** operation on the object, and whatever type of object it has, the correct operation is performed, and the x position is returned.

In the following discussion we use another idiom adopted from the Smalltalk language: performing a generic operation is called *sending a message*. The message consists of an operation name (a symbol) and arguments. One can imagine objects in the program as 'little people' who accept messages and respond to them with answers (returned values). In the example above, an object is sent an **x-position** message, to which it responds with its x position.

Sending a message is a way of invoking a function without specifying which function is to be called. Instead, the data determines the function to use. The caller specifies an operation name and an object; that is, it said what operation to perform, and what object to

perform it on. The function to invoke is found from this information.

The two data used to figure out which function to call are the *type* of the object, and the *name* of the operation. The same set of functions are used for all instances of a given type, so the type is the only attribute of the object used to figure out which function to call. The rest of the message besides the operation is data which are passed as arguments to the function, so the operation is the only part of the message used to find the function. Such a function is called a *method.* For example, if we send an **x-position** message to an object of type ship, then the function we find is *the* ship *type's* **x-position** *method.* A method is a function that handles a specific operation on a specific kind of object; this method handles messages named **x-position** to objects of type ship.

In our new terminology: the orbit-calculating program finds the x position of the object it is working on by sending that object a message consisting of the operation x-position and no arguments. The returned value of the message is the x position of the object. If the object was of type ship, then the ship type's **x-position** method was invoked; if it was of type meteor, then the meteor type's **x-position** method was invoked. The orbit-calculating program just sends the message, and the right function is invoked based on the type of the object. We now have true generic functions, in the form of message passing: the same operation can mean different things depending on the type of the object.

6.5 Generic operations in LISP

How do we implement message passing in LISP? Our convention is that objects that receive messages are always *functional* objects (that is, you can apply them to arguments). A message is sent to an object by calling that object as a function, passing the operation name as the first argument and the arguments of the message as the rest of the arguments. Operation names are represented by symbols; normally these symbols are in the keyword package, since messages may normally be passed between objects defined in different packages. So if we have a variable my-ship whose value is an object of type ship, and we want to know its x position, we send it a message as follows:

```
(send my-ship :x-position)
```

To set the ship's x position to *3.0*, we send it a message like this:

```
(send my-ship :set-x-position 3.0)
```

A variation supported in some Flavor systems would allow

```
(send my-ship :set :x-position 3.0)
;;; not supported
```

but this is now deprecated and not provided in Tek COMMON LISP.

It should be stressed that no new features are added to LISP for message sending; we simply define a convention on the way objects take arguments. The convention says that an object accepts messages by always interpreting its first argument as an operation name. The object must consider this operation name, find the function which is the method for that operation, and invoke that function.

To emphasize the relationship between well-known features and the new object-oriented version, we define the two basic functions for message passing as follows:

send object message ***&rest*** arguments *[Function]*

■ This function is equivalent to **funcall**; however, **send** may be more efficient in some implementations because **funcall** must determine the type of object it is passed, whereas **send** can assume that object is a flavor instance. In any case, the function **send** is preferable to **funcall** when a message is being sent, since it documents that Flavors and message sending are being used.

☐ Conceptually, this sends object a message with operation and arguments as specified.

☐ In some implementations of Flavors, the semantics of **send** may differ from **funcall** in those cases where object is a symbol, list, number, or other object that does not normally handle messages.

lexpr-send object message {arguments}* list-of-arguments *[Macro]*

■ This function is equivalent to **apply**; see the notes above for **send**. The last argument should be a list.

How does this all work? The object must somehow find the right method for the message it is sent. Furthermore, the object now has to be callable as a function. However, an ordinary function will not do: we need a data structure that can store the instance variables (the internal state) of the object. Of the Tek COMMON LISP features available, the most appropriate is the closure. A message-receiving object could be implemented as a closure over a set of instance variables. The function inside the closure would have a big **case** form to dispatch on its first argument.

While using closures would work, it has several problems. The main problem is that in order to add a new operation to a system, it is necessary to modify code in more than one place: you have to find all

the types that *understand* that operation, and add a new clause to the **case**. The problem with this is that you cannot textually separate the implementation of your new operation from the rest of the system: the methods must be interleaved with the other operations for the type. Adding a new operation should only require *adding* LISP code; it should not require *modifying* LISP code.

For example, the conventional way of making generic operations for arithmetic on various new mathematical objects is to have a procedure for each operation (**+**, *****, etc), which has a big **case** for all the types; this means you have to modify code in **generic-plus**, **generic-times**, ... to add a type. This is inconvenient and error-prone.

The *flavor* mechanism is a streamlined, more convenient, and time-tested system for creating message-receiving objects. With flavors, you can add a new method simply by adding code, without modifying existing code. Furthermore, many common and useful things are very easy to do with flavors. The rest of this chapter describes flavors.

6.6 Simple use of flavors

A *flavor*, in its simplest form, is a definition of an abstract type. New flavors are created with the **defflavor** special form, and methods of the flavor are created with the **defmethod** special form. New instances of a flavor are created with the **make-instance** function. This section explains simple uses of these forms.

For an example of a simple use of flavors, here is how the ship example above would be implemented.

```
(defflavor ship (x-position
                 y-position
                 x-velocity
                 y-velocity
                 mass)
                ()
                :gettable-instance-variables)

(defmethod (ship :speed) ()
   (sqrt (+ (expt x-velocity 2)
            (expt y-velocity 2))))

(defmethod (ship :direction) ()
   (atan y-velocity x-velocity))
```

The code above creates a new flavor. The first subform of the **defflavor** is ship, which is the name of the new flavor. Next is the list of instance variables; they are the five that should be familiar by now. The next subform is something we will get to later. The rest of

the subforms are the body of the **defflavor**, and each one specifies an option about this flavor. In our example, there is only one option, namely ***:gettable-instance-variables***. This means that for each instance variable, a method should automatically be generated to return the value of that instance variable. The name of the operation is a symbol with the same name as the instance variable, but interned in the keyword package. Thus, methods are created to handle the operations ***:x-position***, ***:y-position***, and so on.

Each of the two **defmethod** forms adds a method to the flavor. The first one adds a handler to the flavor ship for the operation ***:speed***. The second subform is the lambda-list, and the rest is the body of the function that handles the ***:speed*** operation. The body can refer to or set any instance variables of the flavor, just like variables bound by a containing **let**. When any instance of the ship flavor is invoked with a first argument of ***:direction***, the body of the second **defmethod** is evaluated in an environment in which the instance variables of ship refer to the instance variables of this instance (the one to which the message was sent). So the arguments passed to **atan** are the the velocity components of this particular ship. The result of **atan** becomes the value returned by the ***:direction*** operation.

Now we have seen how to create a new abstract type: a new flavor. Every instance of this flavor has the five instance variables named in the **defflavor** form, and the seven methods we have seen (five that were automatically generated because of the ***:gettable-instance-variables*** option, and two that we wrote ourselves). The way to create an instance of our new flavor is with the **make-instance** function. Here is how it could be used:

```
(setq my-ship (make-instance 'ship))
```

This returns an object whose printed representation is something like *#<ship 13731210>*. (The details of the print form will vary; it is an object which cannot be read back in from this default short-hand printed representation.) The argument to **make-instance** is the name of the flavor to be instantiated. Additional arguments, not used here, are *init options*, that is, commands to the flavor of which we are making an instance, selecting optional features. This will be discussed more in a moment.

Examination of the flavor we have defined shows that it is quite useless as it stands, since there is no way to set any of the parameters. We can fix this up easily by putting the ***:settable-instance-variables*** option into the **defflavor** form. This option tells **defflavor** to generate methods for operations ***:set-x-position***, ***:set-y-position***, and so on. Each such method takes one argument and sets the corresponding instance variable to that value.

Another option we can add to the **defflavor** is ***:initable-instance-variables***, (alternative spelling for compatibility is ***:inittable-instance-variables***) which allows us to initialize the values of the instance variables when an instance is first created. ***:initable-instance-variables*** does not create any methods; instead, it makes *initialization keywords* named ***:x-position***, ***:y-position***, etc., that can be used as init-option arguments to **make-instance** to initialize the corresponding instance variables. The list of init options is sometimes called the *init-plist* because it is like a property list.

Here is the improved **defflavor**:

```
(defflavor ship (x-position
                 y-position
                 x-velocity
                 y-velocity
                 mass)
                ()
                :gettable-instance-variables
                :settable-instance-variables
                :initable-instance-variables)
```

All we have to do is evaluate this new **defflavor**, and the existing flavor definition is updated and now includes the new methods and initialization options. In fact, the instance we generated a while ago now accepts the new operations! We can set the mass of the ship we created by evaluating:

```
(send my-ship :set-mass 3.0)
```

and the mass instance variable of my-ship is properly set to *3.0*.

If you want to play around with flavors, it is useful to know that **describe** of an instance tells you the flavor of the instance and the values of its instance variables. If we were to evaluate (describe my-ship) at this point, the following would be printed:

```
#<ship 3214320>, an object of flavor ship,
 has instance variable values:
    x-position:          nil
    y-position:          nil
    x-velocity:          nil
    y-velocity:          nil
    mass:                3.0
```

Now that the instance variables are *initable*, we can create another ship and initialize some of the instance variables using the init-plist. Let's do that and **describe** the result:

```
=> (setq her-ship
         (make-instance 'ship
                        :x-position 0.0
                        :y-position 2.0
                        :mass 3.5))

#<ship 3242340>

=> (describe her-ship)
#<ship 3242340>, an object of flavor ship,
 has instance variable values:
    x-position:          0.0
    y-position:          2.0
    x-velocity:          nil
    y-velocity:          nil
    mass:                3.5
```

A flavor can also establish default initial values for instance variables. These default values are used when a new instance is created if the values are not initialized any other way. The syntax for specifying a default initial value is to replace the name of the instance variable by a list, whose first element is the name and whose second is a form to evaluate to produce the default initial value. For example when read in the definitions:

```
(defvar *default-x-velocity* 2.0)
(defvar *default-y-velocity* 3.0)

(defflavor ship ((x-position 0.0)
                 (y-position 0.0)
                 (x-velocity *default-x-velocity*)
                 (y-velocity *default-y-velocity*)
                 mass)
                ()
                :gettable-instance-variables
                :settable-instance-variables
                :initable-instance-variables)
```

Then the results are as follows:

```
=> (setq another-ship (make-instance 'ship :x-
position 3.4))
#<ship 2342340>
=> (describe another-ship)
#<ship 2342340>
an object of flavor ship,
 has instance variable values:
    x-position:          3.4
    y-position:          0.0
    x-velocity:          2.0
```

```
y-velocity:          3.0
mass:                nil
```

The value of x-position was initialized explicitly, so the default was ignored. The value of y-position was initialized from the default value, which was *0.0*. The two velocity instance variables were initialized from their default values, which came from two global variables. The value of mass was not explicitly initialized and did not have a default initialization, so it was left as *nil*. Some flavor implementations set an uninitialized instance variable to **unbound** rather than *nil*.

There are many other options that can be used in **defflavor**, and the init options can be used more flexibly than just to initialize instance variables; full details are given later in this chapter. But even with the small set of features we have seen so far, it is easy to write object-oriented programs.

6.7 Mixing flavors

Now we have a system for defining message-receiving objects so that we can have generic operations. If we want to create a new type called meteor that would accept the same generic operations as ship, we could simply write another **defflavor** and two more **defmethod**s that looked just like those of ship, and then meteors and ships would both accept the same operations. Objects of type ship would have some more instance variables for holding attributes specific to ships and some more methods for operations that are not generic, but are only defined for ships; the same would be true of meteor.

However, this would be a a wasteful thing to do. The same code has to be repeated in several places, and several instance variables have to be repeated. The code now needs to be maintained in many places, which is always undesirable. The power of flavors (and the name *flavors*) comes from the ability to mix several flavors and get a new flavor. Since the functionality of ship and meteor partially overlap, we can take the common functionality and move it into its own flavor, which might be called moving-object. We would define moving-object the same way as we defined ship in the previous section. Then, ship and meteor could be defined like this:

```
(defflavor ship (engine-power
                 number-of-passengers
                 name)
                (moving-object)
                :gettable-instance-variables)

(defflavor meteor (percent-iron)
                  (moving-object)
```

```
:initable-instance-variables)
```

These **defflavor** forms use the second subform, for which we previously used (). The second subform is a list of flavors to be combined to form the new flavor; such flavors are called *components*. Concentrating on ship for a moment (analogous statements are true of meteor), we see that it has exactly one component flavor: moving-object. It also has a list of instance variables, which includes only the ship-specific instance variables and not the ones that it shares with meteor. By incorporating moving-object, the ship flavor acquires all of its instance variables, and so need not name them again. It also acquires all of moving-objects methods, too. So with the new definition, ship instances still implement the ***:x-velocity*** and ***:speed*** operations, with the same meaning as before. However, the ***:engine-power*** operation is also understood (and returns the value of the engine-power instance variable).

What we have done here is to take an abstract type, moving-object, and build two more specialized and powerful abstract types on top of it. Any ship or meteor can do anything a moving object can do, and each also has its own specific abilities. This kind of building can continue; we could define a flavor called ship-with-passenger that was built on top of ship, and it would inherit all of moving-objects instance variables and methods as well as ships instance variables and methods. Furthermore, the second subform of **defflavor** can be a list of several components, meaning that the new flavor should combine all the instance variables and methods of all the flavors in the list, as well as the ones *those* flavors are built on, and so on. All the components taken together form a big tree of flavors. A flavor is built from its components, its components' components, and so on. We sometimes use the term *components* to mean the immediate components (the ones listed in the **defflavor**), and sometimes to mean all the components (including the components of the immediate components and so on). (Actually, it is not strictly a tree, since some flavors might be components through more than one path. It is really a directed graph; it can even be cyclic.)

The order in which the components are combined to form a flavor is important. The tree of flavors is turned into an ordered list by performing a *top-down, depth-first* walk of the tree, including non-terminal nodes *before* the subtrees they head, ignoring any flavor that has been encountered previously somewhere else in the tree. For example, if flavor-1s immediate components are flavor-2 and flavor-3, and flavor-2s components are flavor-4 and flavor-5, and flavor-3s component was flavor-4, then the complete list of components of flavor-1 would be: *(flavor-1, flavor-2, flavor-4, flavor-5, flavor-3)*. The flavors earlier in this list are the more specific, less basic ones; in our

example, ship-with-passengers would be first in the list, followed by ship, followed by moving-object. A flavor is always the first in the list of its own components. Notice that flavor-4 does not appear twice in this list. Only the first occurrence of a flavor appears; duplicates are removed. (The elimination of duplicates is done during the walk; a cycle in the directed graph does not cause a non-terminating computation.)

The set of instance variables for the new flavor is the union of all the sets of instance variables in all the component flavors. If both flavor-2 and flavor-3 have instance variables named foo, then flavor-1 has an instance variable named foo, and all methods that refer to foo refer to this same instance variable. Thus different components of a flavor can communicate with one another using shared instance variables. (Often, only one component ever sets the variable; the others only look at it.) The default initial value for an instance variable comes from the first component flavor to specify one.

The way the methods of the components are combined is the heart of the flavor system. When a flavor is defined, a single function, called a *combined method*, is constructed for each operation supported by the flavor. This function is constructed out of all the methods for that operation from all the components of the flavor. There are many different ways that methods can be combined; these can be selected by the user when a flavor is defined. The user can also create new forms of combination.

There are several kinds of methods, but so far, the only kinds of methods we have seen are *primary* methods. The default way primary methods are combined is that all but the earliest one provided are ignored. In other words, the combined method is simply the primary method of the first flavor to provide a primary method. What this means is that if you are starting with a flavor foo and building a flavor bar on top of it, then you can override foos method for an operation by providing your own method. Your method will be called, and foo's will never be called.

Simple overriding is often useful; for example, if you want to make a new flavor bar that is just like foo except that it reacts completely differently to a few operations. However, often you don't want to completely override the base flavor's (foo's) method; sometimes you want to add some extra things to be done. This is where combination of methods is used.

The usual way methods are combined is that one flavor provides a primary method, and other flavors provide *daemon methods*. The idea is that the primary method is *in charge* of the main business of handling the operation, but other flavors just want to keep informed that the message was sent, or just want to do the part of the operation associated with their own area of responsibility.

daemon methods come in two kinds, *before* and *after*. There is a special syntax in **defmethod** for defining such methods. Here is an example of the syntax. To give the ship flavor an after-daemon method for the ***:speed*** operation, the following syntax would be used:

```
(defmethod (ship :after :speed) ( body))
```

Now, when a message is sent, it is handled by a new function called the *combined* method. The combined method first calls all of the before daemons, then the primary method, then all the after daemons. Each method is passed the same arguments that the combined method was given. The returned values from the combined method are the values returned by the primary method; any values returned from the daemons are ignored. Before-daemons are called in the order that flavors are combined, while after-daemons are called in the reverse order. In other words, if you build bar on top of foo, then bar's before-daemons run before any of those in foo, and bars after-daemons run after any of those in foo.

The reason for this order is to keep the modularity order correct. If we create flavor-1 built on flavor-2, then the components of flavor-2 should not matter. Our new before-daemons go before all methods of flavor-2, and our new after-daemons go after all methods of flavor-2. Note that if you have no daemons, this reduces to the form of combination described above. The most recently added component flavor is the highest level of abstraction; you build a higher-level object on top of a lower-level object by adding new components to the front. The syntax for defining daemon methods can be found in the description of **defmethod** below.

To make this a bit more clear, let's consider a simple example that is easy to play with: the ***:print-self*** method. The LISP printer (i.e. the **print** function) prints instances of flavors by sending them ***:print-self*** messages. The first argument to the ***:print-self*** operation is a port (we can ignore the others for now), and the receiver of the message is supposed to print its printed representation on the port. In the ship example above, the reason that instances of the ship flavor printed the way they did is because the ship flavor was actually built on top of a very basic flavor called vanilla-flavor; this component is provided automatically by **defflavor**. It was vanilla-flavor's ***:print-self*** method that was doing the printing. Now, if we give ship its own primary method for the ***:print-self*** operation, then that method completely takes over the job of printing: vanilla-flavors method will not be called at all. However, if we give ship a before-daemon method for the ***:print-self*** operation, then it will get invoked before the vanilla-flavor method, and so whatever it prints will appear before what vanilla-flavor prints. So we can use before-daemons to add

prefixes to a printed representation; similarly, after-daemons can add suffixes.

There are other ways to combine methods besides daemons, but this way is the most common. The more advanced ways of combining methods are explained in a later section. The details of vanilla-flavor and what it does for you are also explained later.

6.8 Flavor functions

We've been using the following special form informally:

defflavor flavor-name ({vars}*) ({flavors}*) {options}* *[Macro]*

■ WHERE flavor-name is a symbol which serves to name this flavor.

☐ The vars are the names of the instance-variables containing the local state for this flavor. A list of two elements: the name of an instance-variable and a default initialization form is also acceptable; the initialization form is evaluated when an instance of the flavor is created if no other initial value for the variable is obtained. If no initialization is specified, the variable has value *nil*.

☐ The flavors are the names of the component flavors out of which this flavor is built. The features of those flavors are inherited as described previously.

☐ Each of the options may be either a keyword symbol or a list of a keyword symbol and arguments. The options to **defflavor** are described under *Defflavor Options*, below.

☐ **type-of** applied to an instance returns the symbol which is the name of its flavor.

■ SIDE EFFECT: The symbol flavor-name is given a flavor property which is the internal data-structure containing the details of the flavor.

■ NOTE: In Tek COMMON LISP objects which are instances of flavors are implemented by a hidden internal data type, actually a kind of vector. The **svref** function can access the slots of an instance. The zeroth slot points to the internal descriptor for that flavor; successive slots hold the instance variables.

all-flavor-names *[Variable]*

■ A special variable containing a list of the names of all flavors that have ever been **defflavor**ed.

defmethod (flavor-name [method-type] operation) *[Macro]*
lambda-list {forms}*

■ WHERE flavor-name is a symbol which is the name of the flavor which is to receive the method. operation is a keyword symbol which names the operation to be handled. method-type is a keyword symbol for the type of method; it is omitted when you are defining a primary method. For some method-types, additional information is expected. It comes after operation.

■ SIDE EFFECT: **defmethod** defines a method, that is, a function to handle a particular operation for instances of a particular flavor. The meaning of method-type depends on what style of method combination is declared for this operation. For instance, if ***:daemon*** combination (the default style) is in use, method types ***:before*** and ***:after*** are allowed. See the section below on *Method Combination* for a complete description of the way methods are combined.

☐ lambda-list describes the arguments and ***&aux*** variables of the function. The first argument to the method, which is the operation name itself, is automatically handled and so is not included in lambda-list. Note that methods may not have unevaluated arguments; that is, they must be functions, not macros or special forms. The forms are the function body; the value of the last form is returned when the method is applied. Some methods can return multiple values, depending on the style of method combination used.

☐ The variant form

```
(defmethod (flavor-name operation) function)
```

where function is a symbol, says that flavor-names method for operation is function, a symbol which names a function. When function is called, self and any special instance variables will be bound. The function must take appropriate arguments; the first argument is the operation. Various flavor implementations have different conventions for automatically-supplied arguments to method functions; these should be conditionalized if code must be transportable.

If you redefine a method that is already defined, the new definition replaces the old one. Given a flavor, an operation name, and a method type, there can only be one function (with the exception of ***:case*** methods), so if you define a ***:before*** daemon method for the foo flavor to handle the ***:bar*** operation, then you replace the previous before-daemon; however, you do not affect the primary method or methods of any other type, operation or flavor.

Along with other things, **defmethod** causes a function to be **defuned**. The function name is formed by concatenating the hyphen-separated print names of all the symbols in the first **defmethod** subform, then suffixing *-method*; this name is interned in the same package as the flavor name. For example, (defmethod (foo :before :bar) ...) defines a function named **foo-before-bar-method**. This is useful to know if you want to trace a method, or if you want to poke around at the method function itself.

make-instance flavor-name {init-option value}* *[Function]*

■ RETURNS an instance of the specified flavor which has just been created.

□ Arguments after the first are alternating init-option keywords and arguments to those keywords. These options are used to initialize instance variables and to select arbitrary options, as described above. An ***:init*** message is sent to the newly-created object with one argument, the init-plist. This is a property-list containing the init-options specified and those defaulted from the flavor's ***:default-init-plist*** (however, init keywords that simply initialize instance variables, and the corresponding values, may be absent when the ***:init*** methods are called). **make-instance** is an easy-to-call interface to **instantiate-flavor**, below.

instantiate-flavor flavor-name init-plist ***&optional*** send-init-message-p return-unhandled-keywords area *[Function]*

■ RETURNS a new instance of flavor flavor-name.

■ NOTE: This is an extended version of **make-instance**, giving you more features. Note that it takes the init-plist as an individual argument, rather than taking a ***&rest*** argument of init options and values.

This property list can be modified; the properties from the default init-plist are added on if not already present, and some ***:init*** methods may do explicit (setf (getf ...)) onto the init-plist.

In the event that ***:init*** methods **remprop** properties already on the init-plist, as opposed to simply doing (setf (getf ...)), then the init-plist is **rplacd**ed. This means that the actual supplied list of options is modified, so this list should not be one contained inside a body of code. This would permanently modify the calling code. Therefore for each call of **instantiate-flavor** the caller should recreate or otherwise copy (e.g. with **append**) the list to be passed as the init-plist argument.

Do not use *nil* as the init-plist argument. This would mean to use the properties of the symbol *nil* as the init options. If your goal is to

have no init options, you must provide a property list containing no properties, such as the list (nil), which can be created by evaluating the form (list nil).

Here is the sequence of actions by which **instantiate-flavor** creates a new instance:

(1) The specified flavor's instantiation flavor function if it exists, is called to determine which flavor should actually be instantiated. If there is no instantiation flavor function, the specified flavor is instantiated.
If the flavor's method hash-table and other internal information have not been computed or are not up to date, they are computed. This may take a substantial amount of time, but it happens only once for each time you define or redefine a particular flavor.

(2) The instance itself is created. The area argument is ignored by Tek COMMON LISP and refers to consing in specified areas, a feature used in some LISP machines.

(3) Initial values of the instance variables are computed. If an instance variable is declared initable, and a keyword with the same spelling as its name appears in init-plist, the property for that keyword is used as the initial value.
Otherwise, if the default init-plist specifies such a property, the value form is evaluated and the result used. Otherwise, if the flavor definition specifies a default initialization form, it is evaluated and that result is used. In either case, the initialization forms may not refer to any instance variables, nor will they find the variable self be bound to the new instance. The value forms are evaluated before the instance is actually allocated.
If an instance variable does not get initialized either of these ways it is left *nil*; an ***:init*** method may initialize it (see below).
All remaining keywords and values specified in the ***:default-init-plist*** option to **defflavor**, that do not initialize instance variables and are not overridden by anything explicitly specified in init-plist are then merged into init-plist using **setf** of **getf**. The default init plist of the instantiated flavor is considered first, followed by those of all the component flavors in the standard order.

(4) Keywords appearing in the init-plist but not defined with the ***:init-keywords*** option or the ***:initable-instance-variables*** option for some component flavor are collected. If the ***:allow-other-keys*** option is specified with a non-*nil* value (either in the original init-plist argument or by some default init plist) then these *unhandled* keywords are ignored. If the return-unhandled-keywords argument is non-*nil*, a list of these keywords is returned as the second value of **instantiate-flavor**. Otherwise, an error is signaled if any unrecognized init keywords are present.

(5) If the send-init-message-p argument is supplied and non-*nil*, an ***:init*** message is sent to the newly-created instance, with one argument, the init-plist. **getf** can be used to extract options from this property-list. Each flavor that needs initialization can contribute an ***:init*** method by defining a daemon.

The ***:init*** methods should not look on the init-plist for keywords that simply initialize instance variables (that is, keywords defined with ***:initable-instance-variables*** rather than ***:init-keywords***). The corresponding instance variables are already set up when the :init methods are called, and sometimes the keywords and their values may actually be missing from the init-plist if it is more efficient not to put them on. To avoid problems, always refer to the instance variables themselves rather than looking for the init keywords that initialize them.

:init init-plist *[Message]*

■ This operation is implemented on all flavor instances.

■ SIDE EFFECT: This function examines the init keywords and perform whatever initializations are appropriate. init-plist is the argument that was given to **instantiate-flavor**, and may be passed directly to **getf** to examine the value of any particular init option.

□ The default definition of this operation does nothing. However, many flavors add ***:before*** and ***:after*** daemons to it.

instancep object *[Function]*

■ RETURNS *t* if object is an instance of a flavor.

defwrapper (flavor-name operation) lambda-list ***&body*** body *[Macro]*

■ NOTE: This feature is complex and you may not be able to understand it completely until you have gained some experience with flavors. It can safely be skipped meanwhile.

□ Sometimes the way the flavor system combines the methods of different flavors (the daemon system) is not powerful enough. In that case **defwrapper** can be used to define a macro that expands into code that is wrapped around the invocation of the methods. This is best explained by an example; suppose you needed a lock locked during the processing of the ***:foo*** operation on flavor bar, which takes two arguments, and you have a lock-frobboz special-form that knows how to lock the lock (presumably it generates an **unwind-protect**). lock-frobboz needs to see the first argument to the operation; perhaps that tells it what sort

of operation is going to be performed (read or write).

```
(defwrapper (bar :foo) ((arg1 arg2) . body)
  `(lock-frobboz (self arg1)
     . ,body))
```

The use of the body macro-argument prevents the macro defined by **defwrapper** from knowing the exact implementation and allows several **defwrapper**s from different flavors to be combined properly.

Note that the argument variables, arg1 and arg2, are not referenced with commas before them. These may look like **defmacro** argument variables, but they are not. Those variables are not bound at the time the **defwrapper**-defined macro is expanded and the back-quoting is done; rather, the result of that macro-expansion and back-quoting is code which, when a message is sent, will bind those variables to the arguments in the message as local variables of the combined method.

Consider another example. Suppose you thought you wanted a ***:before*** daemon, but found that if the argument was *nil* you needed to return from processing the message immediately, without executing the primary method. You could write a wrapper such as:

```
(defwrapper (bar :foo) ((arg1) . body)
  `(cond ((null arg1))
         (t (print "About to do :FOO")
            . ,body)))
```

Suppose you need a variable for communication among the daemons for a particular operation; perhaps the ***:after*** daemons need to know what the primary method did, and it is something that cannot be easily deduced from just the arguments. You might use an instance variable for this, or you might create a special variable which is bound during the processing of the operation and used free by the methods.

```
(defvar *communication*)
(defwrapper (bar :foo) (ignore . body)
  `(let ((*communication* nil))
     . ,body))
```

Similarly you might want a wrapper that puts a **catch** around the processing of an operation so that any one of the methods could **throw** out in the event of an unexpected condition.

Like daemon methods, wrappers work in outside-in order; when you add a **defwrapper** to a flavor built on other flavors, the new wrapper is placed outside any wrappers of the component

flavors. However, *all* wrappers happen before *any* daemons happen. When the combined method is built, the calls to the before-daemon methods, primary methods, and after-daemon methods are all placed together, and then the wrappers are wrapped around them. Thus, if a component flavor defines a wrapper, methods added by new flavors execute within that wrapper's context.

Be careful about inserting the body into an internal lambda-expression within the wrapper's code. Doing so interacts with the internals of the flavor system and requires knowledge of things not documented in the manual in order to work properly.

defwhopper (flavor-name operation) lambda-list ***&body*** body *[Macro]*

■ NOTE: Whoppers are a feature of some flavor implementations which do many of the same things as wrappers. They will be documented when they are implemented in Tek COMMON LISP.

undefmethod flavor [type] operation [suboperation] *[Macro]*

■ Removes a method: (undefmethod (flavor :before :operation)) removes the method created by (defmethod (flavor :before :operation) ...). To remove a wrapper, use **undefmethod** with ***:wrapper*** as the method type.

undefflavor flavor *[Function]*

■ Undefines flavor flavor. All methods of the flavor are lost. flavor and all flavors that depend on it are no longer valid to instantiate. If instances of the discarded definition exist, they continue to use that definition.

self *[Variable]*

■ When a message is sent to an object, the variable *self* is automatically bound to that object for the benefit of methods which want to manipulate the object itself (as opposed to its instance variables). *self* is a lexical variable, that is, its scope of is local to the method body.

send instance message [argument ...] *[Macro]*
funcall instance message ***&rest*** arguments *[Function]*

■ NOTE: This is the way a message is passed to an instance of a flavor. **send** and **funcall** operate in essentially the same manner. **send** is potentially slightly more efficient because the evaluator can infer that the functional argument is an instance, whereas

funcall must determine the type of its first argument.

send-self message {arguments}* *[Macro]*
funcall-self message {arguments}* *[Macro]*
lexpr-send-self message {arguments}* list-of-arguments *[Macro]*
lexpr-funcall-self message {arguments}* list-of-arguments *[Macro]*

■ **funcall-self** is nearly equivalent to **funcall** with **self** as the first argument, but may be a little faster. The others are analogous.

recompile-flavor flavor-name &optional single-op use-old-combined-methods do-dependents *[Function]*

■ Updates the internal data of the flavor and any flavors that depend on it. If single-op is supplied non-*nil*, only the methods for that operation are changed. The system does this when you define a new method that did not previously exist. If use-old-combined-methods is *t*, then the existing combined method functions are used if possible. New ones are generated only if the set of methods to be called has changed. This is the default. If use-old-combined-methods is *nil*, automatically-generated functions to call multiple methods or to contain code generated by wrappers are regenerated unconditionally. If do-dependents is *nil*, only the specific flavor you specified is recompiled. Normally all flavors that depend on it are also recompiled.
□ **recompile-flavor** affects only flavors that have already been compiled. Typically this means it affects flavors that have been instantiated, but does not bother with mixins.

compile-flavor-methods {flavor-names}* *[Macro]*

■ The form (compile-flavor-methods flavor-name-1 flavor-name-2 ...), placed in a file to be compiled, directs the compiler to perform flavor combination for the named flavors, forcing the generation and compilation of automatically-generated combined methods at compile time. Furthermore, the internal data structures needed to instantiate the flavor will be computed at load time, rather than waiting for the first attempt to instantiate the flavor.
□ You should only use **compile-flavor-methods** on a flavor that is going to be instantiated. For a flavor that is never going to be instantiated (that is, a flavor that only serves to be a component of other flavors that actually do get instantiated), it is a

complete waste of time, except in the unusual case where those other flavors can inherit the combined methods of this flavor instead of each one having its own copy of the combined method which happens to identical to the others. In this unusual case, you should use the ***:abstract-flavor*** option to **defflavor**.

☐ **compile-flavor-methods** forms should be compiled after all of the other information needed to create the combined methods is available. You should put them after all the definitions of all relevant flavors, wrappers, and methods of all components of the argument flavors.

☐ When a **compile-flavor-methods** form is seen by the interpreter, the internal data structures are generated and the combined methods are defined.

get-handler-for object operation *[Function]*

■ Given an object and an operation, this returns the object's method for that operation, or *nil* if it has none. When object is an instance of a flavor, this function can be useful to find which of that flavor's components supplies the method.

☐ This is equivalent to the ***:get-handler-for*** message provided by ***si:vanilla-flavor***.

flavor-allows-init-keyword-p flavor-name keyword *[Function]*

■ RETURNS non-*nil* if the flavor named flavor-name allows keyword in the init options when it is instantiated, or *nil* if it does not. The non-*nil* value is the name of the component flavor that contributes the support of that keyword.

si:flavor-allowed-init-keywords flavor-name *[Function]*

■ RETURNS a list of all the init keywords that may be used in instantiating flavor-name.

symeval-in-instance instance symbol ***&optional*** no-error-p *[Function]*

■ RETURNS the value of the instance variable symbol inside instance. If there is no such instance variable, an error is signaled, unless no-error-p is non-*nil*, in which case *nil* is returned.

set-in-instance instance symbol value *[Function]*

■ SIDE EFFECT: Sets the value of the instance variable symbol inside instance to value. If there is no such instance variable, an error is signaled.

describe-flavor flavor-name *[Function]*

■ SIDE EFFECT: Prints descriptive information about a flavor; it is self-explanatory. An important thing it tells you that can be hard to figure out yourself is the combined list of component flavors; this list is what is printed after the phrase 'and directly or indirectly depends on.'

6.9 Defflavor options

There are quite a few options to **defflavor**. They are all described here, although some are for very specialized purposes and not of interest to most users. A few options take additional arguments, and these are listed and described with the option.

Several of these options declare things about instance variables. These options can be given with arguments which are instance variables, or without any arguments in which case they refer to all of the instance variables listed at the top of the **defflavor**. This is *not* necessarily all the instance variables of the combined flavor, just the ones mentioned in this flavor's **defflavor**. When arguments are given, they must be instance variables that were listed at the top of the **defflavor**; otherwise they are assumed to be misspelled and an error is signaled. It is legal to declare things about instance variables inherited from a component flavor, but to do so you must list these instance variables explicitly in the instance variable list at the top of the **defflavor**, or mention them in a **required-instance-variable** option.

:gettable-instance-variables *[Defflavor option]*

■ Enables automatic generation of methods for getting the values of instance variables. The operation name is the name of the variable, in the keyword package (i.e. it has a colon in front of it).

□ Note that there is nothing special about these methods; you could easily define them yourself. This option generates them automatically to save you the trouble of writing out a lot of very simple method definitions. (The same is true of methods defined by the :settable-instance-variables option.) If you define a method for the same operation name as one of the automatically generated methods, the explicit definition overrides the automatic one.

:settable-instance-variables *[Defflavor option]*

■ Enables automatic generation of methods for setting the values of instance variables. The operation name is ***:set-*** followed by

the name of the variable. All settable instance variables are also automatically made gettable and initable. (See the note in the description of the ***:gettable-instance-variables*** option, above.)

:initable-instance-variables *[Defflavor option]*

■ The instance variables listed as arguments, or all instance variables listed in this **defflavor** if the keyword is given alone, are made *initable*. This means that they can be initialized through use of a keyword (a colon followed by the name of the variable) as an init-option argument to **make-instance**. For compatibility with certain other implementations, the spelling ***:inittable-instance-variables*** is also accepted.

:special-instance-variables *[Defflavor option]*

■ NOTE: Special instance variables are not implemented Tek COMMON LISP. Instance variables are scoped lexically inside a method in both compiled and interpreted code. Special instance variables are unimplementable in COMMON LISP for the same reasons that it is impossible to close over a normal special variable. In any case, they are deleterious to proper code modularity; the original designers of Flavors now deprecate them as a misfeature except for very obscure (or historical) purposes. The Tek COMMON LISP implementation ignores the :special-instance-variable specification other than issuing a warning message, but the resulting code will be unlikely to do the right thing if the instance variables were declared special for some particular purpose.

:init-keywords *[Defflavor option]*

■ The arguments are declared to be valid keywords to use in **instantiate-flavor** when creating an instance of this flavor (or any flavor containing it). The system uses this for error-checking: before the system sends the ***:init*** message, it makes sure that all the keywords in the init-plist are either initable instance variables or elements of this list. If any is not recognized, an error is signaled. When you write a ***:init*** method that accepts some keywords, they should be listed in the ***:init-keywords*** option of the flavor. If ***:allow-other-keys*** is used as an init keyword with a non-*nil* value, this error check is suppressed. Then unrecognized keywords are simply ignored.

:default-init-plist *[Defflavor option]*

■ The arguments are alternating keywords and value forms, like a property list. When the flavor is instantiated, these properties and values are put into the init-plist unless already present. This allows one component flavor to default an option to another component flavor. The value forms are only evaluated when and if they are used. For example,

```
(:default-init-plist :frob-array
                     (make-array 100))
```

would provide a default *frob array* for any instance for which the user did not provide one explicitly. The following specification prevents errors for unhandled init keywords in all instantiations of this flavor and other flavors that depend on it.

```
(:default-init-plist :allow-other-keys t)
```

:required-init-keywords *[Defflavor option]*

■ The arguments are init keywords which are to be required each time this flavor (or any flavor containing it) is instantiated. An error is signaled if any required init keyword is missing.

:required-instance-variables *[Defflavor option]*

■ Declares that any flavor incorporating this one that is instantiated into an object must contain the specified instance variables. An error occurs if there is an attempt to instantiate a flavor that incorporates this one if it does not have these in its set of instance variables. Note that this option is not one of those that checks the spelling of its arguments in the way described at the start of this section (if it did, it would be useless).

□ Required instance variables may be freely accessed by methods just like normal instance variables. The difference between listing instance variables here and listing them at the front of the **defflavor** is that the latter declares that this flavor *owns* those variables and accepts responsibility for initializing them, while the former declares that this flavor depends on those variables but that some other flavor must be provided to manage them and whatever features they imply.

:required-methods *[Defflavor option]*

■ The arguments are names of operations that any flavor incorporating this one must handle. An error occurs if there is an attempt to instantiate such a flavor and it is lacking a method for

one of these operations. Typically this option appears in the **defflavor** for a base flavor. Usually this is used when a base flavor does a (send self ...) to send itself a message that is not handled by the base flavor itself; the idea is that the base flavor will not be instantiated alone, but only with other components (mixins) that do handle the message. This keyword allows the error of having no handler for the message to be detected when the flavor instantiated or when **compile-flavor-methods** is done, rather than when the missing operation is used.

:required-flavors *[Defflavor option]*

■ The arguments are names of flavors that any flavor incorporating this one must include as components, directly or indirectly. The difference between declaring flavors as required and listing them directly as components at the top of the **defflavor** is that declaring flavors to be required does not make any commitments about where those flavors will appear in the ordered list of components; that is left up to whoever does specify them as components. Declaring a flavor to be required only provides error checking: an attempt to instantiate a flavor that does not include the required flavors as components signals an error. Compare this with ***:required-methods*** and ***:required-instance-variables***.

For an example of the use of required flavors, consider the ship example given earlier, and suppose we want to define a relativity-mixin which increases the mass dependent on the speed. We might write,

```
(defflavor relativity-mixin () (moving-object))
(defmethod (relativity-mixin :mass) ()
  (/ mass (sqrt (- 1
                   (expt (/ (send self :speed)
                            *speed-of-light*)
                         2)))))
```

but this would lose because any flavor that had relativity-mixin as a component would get moving-object right after it in its component list. As a base flavor, moving-object should be last in the list of components so that other components mixed in can replace its methods and so that daemon methods combine in the right order. relativity-mixin has no business changing the order in which flavors are combined, which should be under the control of its caller. For example,

```
(defflavor starship () (relativity-mixin long-
distance-mixin ship))
```

puts moving-object last (inheriting it from ship). So instead of the definition above we write,

```
(defflavor relativity-mixin () () (:required-
flavors moving-object))
```

which allows relativity-mixins methods to access moving-object instance variables such as mass (the rest mass), but does not specify any place for moving-object in the list of components.

It is very common to specify the *base flavor* of a mixin with the ***:required-flavors*** option in this way.

:included-flavors *[Defflavor option]*

■ The arguments are names of flavors to be included in this flavor. The difference between declaring flavors here and declaring them at the top of the **defflavor** is that when component flavors are combined, if an included flavor is not specified as a normal component, it is inserted into the list of components immediately after the last component to include it. Thus included flavors act like defaults. The important thing is that if an included flavor *is* specified as a component, its position in the list of components is completely controlled by that specification, independently of where the flavor that includes it appears in the list.

:included-flavors and ***:required-flavors*** are used in similar ways; it would have been reasonable to use ***:included-flavors*** in the relativity-mixin example above. The difference is that when a flavor is required but not given as a normal component, an error is signaled, but when a flavor is included but not given as a normal component, it is automatically inserted into the list of components at a reasonable place.

:no-vanilla-flavor *[Defflavor option]*

■ Normally when a flavor is instantiated, the special flavor si:vanilla-flavor is included automatically at the end of its list of components. The vanilla flavor provides some default methods for the standard operations which all objects are supposed to understand. These include ***:print-self***, ***:describe***, ***:which-operations***, and several other operations.

□ If any component of a flavor specifies the ***:no-vanilla-flavor*** option, then si:vanilla-flavor is not included in that flavor. This option should not be used casually.

:default-handler *[Defflavor option]*

■ The argument is the name of a function that is to be called to handle any operation for which there is no method. Its arguments are the arguments of the **send** which invoked the operation, including the operation name as the first argument. Whatever values the default handler returns are the values of the operation.

☐ Default handlers can be inherited from component flavors. If a flavor has no other default handler, one is provided which signals an error if a message is sent for which there is no handler.

:ordered-instance-variables *[Defflavor option]*

■ This option is mostly for esoteric internal system uses. The arguments are names of instance variables which must appear first (and in this order) in all instances of this flavor, or any flavor depending on this flavor. This is used for instance variables that are specially known about by other code (e.g. non-LISP) and also in connection with the ***:outside-accessible-instance-variables*** option. If the keyword is given alone, the arguments default to the list of instance variables given at the top of this **defflavor**.

☐ Any number of flavors to be combined together can specify this option. The longest ordered variable list applies, and an error is signaled if any of the other lists do not match its initial elements.

☐ Removing any of the ***:ordered-instance-variables***, or changing their positions in the list, requires that you recompile all methods that use any of the affected instance variables.

:outside-accessible-instance-variables *[Defflavor option]*

■ The arguments are instance variables which are to be accessible from outside of this flavor's methods. A macro is defined which takes an object of this flavor as an argument and returns the value of the instance variable; **setf** may be used to set the value of the instance variable. The name of the macro is the name of the flavor concatenated with a hyphen and the name of the instance variable. These macros are similar to the accessors created by **defstruct**.

☐ This feature works in two different ways, depending on whether or not the instance variable has been declared to have a fixed slot in all instances, via the ***:ordered-instance-variables*** option.

☐ If the variable is not ordered, the position of its value cell in the instance must be computed at run time. This takes noticeable time, possibly more or less than actually sending a message

would take. An error is signaled if the argument to the accessor macro is not an instance or is an instance that does not have an instance variable with the appropriate name. However, there is no error check that the flavor of the instance is the flavor the accessor macro was defined for, or a flavor built upon that flavor. This error check would be too expensive.

☐ If the variable is ordered, the compiler compiles a call to the accessor macro into a primitive (actually a **svref**) which simply accesses that variable's assigned slot by number. No error-checking is performed to make sure that the argument is really an instance, much less that it is of the appropriate type.

☐ **setf** works on these accessor macros to modify the instance variable.

:accessor-prefix *[Defflavor option]*

■ Normally the accessor macro created by the ***:outside-accessible-instance-variables*** option to access the flavor f's instance variable v is named f-v. This option allows something other than the flavor name to be used for the first part of the macro name. Specifying (:accessor-prefix get$) causes it to be named get$v instead.

:alias-flavor *[Defflavor option]*

■ NOTE: ***:alias-flavor*** is presently unimplemented in Tek COMMON LISP.

■ Marks this flavor as being an alias for another flavor. This flavor should have only one component, which is the flavor it is an alias for, and no instance variables or other options. No methods should be defined for it.

☐ The effect of the ***:alias-flavor*** option is that an attempt to instantiate this flavor actually produces an instance of the other flavor. Without this option, it would make an instance of this flavor, which might behave identically to an instance of the other flavor. ***:alias-flavor*** eliminates the need for separate mapping tables, method tables, etc. for this flavor, which becomes truly just another name for its component flavor.

☐ The alias flavor and its base flavor are also equivalent when used as an argument of **subtypep** or as the second argument of **typep**; however, if the alias status of a flavor is changed, you must recompile any code which uses it as the second argument to **typep** in order for such code to function.

☐ ***:alias-flavor*** is mainly useful for changing a flavor's name gracefully.

:abstract-flavor *[Defflavor option]*

■ This option marks the flavor as one that is not supposed to be instantiated (that is, is supposed to be used only as a component of other flavors). An attempt to instantiate the flavor signals an error.

□ It is sometimes useful to do **compile-flavor-methods** on a flavor that is not going to be instantiated, if the combined methods for this flavor will be inherited and shared by many others. ***:abstract-flavor*** tells **compile-flavor-methods** not to complain about missing required flavors, methods or instance variables. Presumably the flavors that depend on this one and actually are instantiated will supply what is lacking.

■ NOTE: ***:abstract-flavor*** is accepted but currently ignored in Tek COMMON LISP.

:method-combination *[Defflavor option]*

■ Specifies the method combination style to be used for certain operations. Each argument to this option is a list (style order operation1 operation2 ...). operation1, operation2, etc. are names of operations whose methods are to be combined in the declared fashion. style is a keyword that specifies a style of combination. order is a keyword whose interpretation is up to style; typically it is either ***:base-flavor-first*** or ***:base-flavor-last***.

□ Any component of a flavor may specify the type of method combination to be used for a particular operation. If no component specifies a style of method combination, then the default style is used, namely ***:daemon***. If more than one component of a flavor specifies the combination style for a given operation, then they must agree on the specification, or else an error is signaled.

:run-time-alternatives defflavor *[Defflavor option]*
:mixture defflavor *[Defflavor option]*

■ A run-time-alternative flavor defines a collection of similar flavors, all built on the same base flavor but having various mixins as well. Instantiation chooses a flavor of the collection at run time based on the init keywords specified, using an automatically generated instantiation flavor function.

□ A simple example would be

```
(defflavor foo () (basic-foo)
  (:run-time-alternatives
   (:big big-foo-mixin))
   (:init-keywords :big))
```

☐ Then (make-instance 'foo :big t) makes an instance of a flavor whose components are big-foo-mixin as well as foo. But (make-instance 'foo) or (make-instance 'foo :big nil) makes an instance of foo itself. The clause (:big big-foo-mixin) in the ***:run-time-alternatives*** says to incorporate big-foo-mixin if ***:big***'s value is *t*, but not if it is *nil*.

☐ There may be several clauses in the ***:run-time-alternatives***. Each one is processed independently. Thus, two keywords ***:big*** and ***:wide*** could independently control two mixins, giving four possibilities.

```
(defflavor foo () (basic-foo)
  (:run-time-alternatives (:big big-foo-mixin)
                          (:wide wide-foo-
mixin))
  (:init-keywords :big))
```

☐ It is possible to test for values other than *t* and *nil*. The clause:

```
(:size (:big big-foo-mixin)
       (:small small-foo-mixin)
       (nil nil))
```

allows the value for the keyword ***:size*** to be ***:big***, ***:small*** or *nil* (or omitted). If it is *nil* or omitted, no mixin is used (that's what the second *nil* means). If it is ***:big*** or ***:small***, an appropriate mixin is used. This kind of clause is distinguished from the simpler kind by having a list as its second element. The values to check for can be anything, but **eq** is used to compare them.

☐ The value of one keyword can control the interpretation of others by nesting clauses within clauses. If an alternative has more than two elements, the additional elements are subclauses which are considered only if that alternative is selected. For example, the clause:

```
(:etherial (t etherial-mixin)
           (nil nil
                (:size (:big big-foo-mixin)
                       (:small small-foo-mixin)
                       (nil nil))))
```

says to consider the ***:size*** keyword only if ***:etherial*** is *nil*.

☐ ***:mixture*** is synonymous with ***:run-time-alternatives***. It exists for compatibility with ZetaLISP or other LISP Machine systems.

:documentation *[Defflavor option]*

■ Specifies the documentation string for the flavor definition. This documentation can be viewed with the **describe-flavor** function.

6.10 Flavor families

The following organization conventions are recommended for programs that use flavors.

A *base flavor* is a flavor that defines a whole family of related flavors, all of which have that base flavor as a component. Typically the base flavor includes things relevant to the whole family, such as instance variables, ***:required-methods*** and ***:required-instance-variables*** declarations, default methods for certain operations, ***:method-combination*** declarations, and documentation on the general protocols and conventions of the family. Some base flavors are complete and can be instantiated, but most cannot be instantiated themselves. They serve as a base upon which to build other flavors. The base flavor for the foo family is often named basic-foo.

A *mixin flavor* is a flavor that defines one particular feature of an object. A mixin cannot be instantiated, because it is not a complete description. Each module or feature of a program is defined as a separate mixin; a usable flavor can be constructed by choosing the mixins for the desired characteristics and combining them, along with the appropriate base flavor. By organizing your flavors this way, you keep separate features in separate flavors, and you can pick and choose among them. Sometimes the order of combining mixins does not matter, but often it does, because the order of flavor combination controls the order in which daemons are invoked and wrappers are wrapped. Such order dependencies should be documented as part of the conventions of the appropriate family of flavors. A mixin flavor that provides the *mumble* feature is often named mumble-mixin.

If you are writing a program that uses someone else's facility, using that facility's flavors and methods, your program may still define its own flavors, in a simple way. The facility provides a base flavor and a set of mixins: the caller can combine these in various ways depending on exactly what it wants, since the facility probably does not provide all possible useful combinations. Even if your private flavor has exactly the same components as a pre-existing flavor, it can still be useful since you can use its ***:default-init-plist*** to select options of its component flavors and you can define one or two methods to customize it *just a little*.

6.11 Vanilla flavor

The operations described in this section are a standard protocol, which all message-receiving objects are assumed to understand. The standard methods that implement this protocol are automatically supplied by the flavor system unless the user specifically tells it not to do so. These methods are associated with the flavor si:vanilla-flavor:

si:vanilla-flavor *[Flavor]*

■ NOTE: For source code compatibility with other implementations, Tek COMMON LISP defines *si:* as an alias for the *system:* package.

■ Unless you specify otherwise (with the ***:no-vanilla-flavor*** option to **defflavor**), every flavor includes the *vanilla* flavor, which has no instance variables but provides some basic useful methods.

:print-self stream prindepth escape-p *[Message]*

■ The object should output its printed-representation to a stream. The printer sends this message when it encounters an instance or an entity. The arguments are the stream, the current depth in list-structure (for comparison with *print-level*), and whether escaping is enabled (a copy of the value of **print-escape**). si:vanilla-flavor ignores the last two arguments and prints something like *#<flavor-name octal-address>*. The *flavor-name* tells you what type of object it is and the *octal-address* allows you to tell different objects apart.

:describe *[Message]*

■ The object should describe itself, printing a description onto the standard output stream. The **describe** function sends this message when it encounters an instance. si:vanilla-flavor outputs in a reasonable format the object, the name of its flavor, and the names and values of its instance-variables. The instance variables are printed in their order within the instance.

:which-operations *[Message]*

■ The object should return a list of the operations it can handle. si:vanilla-flavor generates the list once per flavor and remembers it, minimizing consing and compute-time. If the set of operations handled is changed, this list is regenerated the next time someone asks for it.

:operation-handled-p operation *[Message]*

■ operation is an operation name. The object should return *t* if it has a handler for the specified operation, *nil* if it does not.

:get-handler-for operation *[Message]*

■ operation is an operation name. The object should return the method it uses to handle operation. If it has no handler for that operation, it should return *nil*. This is like the **get-handler-for** function.

:send-if-handles operation {arguments}* *[Message]*

■ operation is an operation name and arguments is a list of arguments for the operation. If the object handles the operation, it should send itself a message with that operation and arguments, and return whatever values that message returns. If it doesn't handle the operation it should just return *nil*.

:eval-inside-yourself form *[Message]*

■ The argument is a form that is evaluated in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to **setq** one of these special variables; the instance variable is modified. This is intended to be used mainly for debugging.

:funcall-inside-yourself function ***&rest*** args *[Message]*

■ function is applied to args in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables. It works to **setq** one of these special variables; the instance variable is modified. This is a way of allowing callers to provide actions to be performed in an environment set up by the instance.

:break *[Message]*

■ **break** is called in an environment in which special variables with the names of the instance variables are bound to the values of the instance variables.

6.12 Method combination

When a flavor has or inherits more than one method for an operation, they must be called in a specific sequence. The flavor system creates a function called a *combined method* which calls all the user-specified methods in the proper order. Invocation of the operation actually calls the combined method, which is responsible for calling the others.

For example, if the flavor foo has components and methods as follows:

```
(defflavor foo () (foo-mixin foo-base))
(defflavor foo-mixin () (bar-mixin))

(defmethod (foo :before :hack) ...)
(defmethod (foo :after :hack) ...)

(defmethod (foo-mixin :before :hack) ...)
(defmethod (foo-mixin :after :hack) ...)

(defmethod (bar-mixin :before :hack) ...)
(defmethod (bar-mixin :hack) ...)

(defmethod (foo-base :hack) ...)
(defmethod (foo-base :after :hack) ...)
```

then the combined method generated looks like this (ignoring many details not related to this issue):

```
(defmethod (foo :combined :hack) (&rest args)
  (apply #'(:method foo :before :hack) args)
  (apply #'(:method foo-mixin :before :hack) args)
  (apply #'(:method bar-mixin :before :hack) args)
  (multiple-value-prog1
      (apply #'(:method bar-mixin :hack) args)
    (apply #'(:method foo-base :after :hack) args)
    (apply #'(:method foo-mixin :after :hack)
args)
    (apply #'(:method foo :after :hack) args)))
```

This example shows the default style of method combination, the one described in the introductory parts of this chapter, called ***:daemon*** combination. Each style of method combination defines which *method types* it allows, and what they mean. ***:daemon*** combination accepts method types ***:before*** and ***:after***, in addition to *untyped* methods; then it creates a combined method which calls all the ***:before*** methods, only one of the untyped methods, and then all the ***:after*** methods, returning the value of the untyped method. The combined method is constructed by a function much like a macro's expander function, and the precise technique used to create the combined method is what gives ***:before*** and ***:after*** their meaning.

Note that the ***:before*** methods are called in the order foo, foo-mixin, bar-mixin and foo-base. (foo-base does not have a ***:before*** method, but if it had one that one would be last.) This is the standard ordering of the components of the flavor foo; since it puts the base flavor last, it is called ***:base-flavor-last*** ordering. The ***:after*** methods are called in the opposite order, in which the base flavor comes first. This is called ***:base-flavor-first*** ordering.

Only one of the untyped methods is used; it is the one that comes first in ***:base-flavor-last*** ordering. An untyped method used in this way is called a *primary* method.

Other styles of method combination define their own method types and have their own ways of combining them. Use of another style of method combination is requested with the ***:method-combination*** option to **defflavor**. Here is an example which uses ***:list*** method combination, a style of combination that allows ***:list*** methods and untyped methods:

```
(defflavor foo () (foo-mixin foo-base))
(defflavor foo-mixin () (bar-mixin))
(defflavor foo-base () ()
  (:method-combination (:list :base-flavor-last
:win)))

(defmethod (foo :list :win) ...)
(defmethod (foo :win) ...)

(defmethod (foo-mixin :list :win) ...)

(defmethod (bar-mixin :list :win) ...)
(defmethod (bar-mixin :win) ...)

(defmethod (foo-base :win) ...)

;; yielding this combined method

(defmethod (foo :combined :win) (&rest args)
  (list (apply #'(:method foo :list :win) args)
        (apply #'(:method foo-mixin :list :win)
args)
        (apply #'(:method bar-mixin :list :win)
args)
        (apply #'(:method foo :win) args)
        (apply #'(:method bar-mixin :win) args)
        (apply #'(:method foo-base :win) args)))
```

The ***:method-combination*** option in the **defflavor** for foo-base causes ***:list*** method combination to be used for the ***:win*** operation on all flavors that have foo-base as a component, including foo. The

result is a combined method which calls all the methods, including all the untyped methods rather than just one, and makes a list of the values they return. All the ***:list*** methods are called first, followed by all the untyped methods; and within each type, the ***:base-flavor-last*** ordering is used as specified. If the ***:method-combination*** option said ***:base-flavor-first***, the relative order of the ***:list*** methods would be reversed, and so would the untyped methods, but the ***:list*** methods would still be called before the untyped ones. ***:base-flavor-last*** is more often right, since it means that foo's own methods are called first and si:vanilla-flavor's methods (if it has any) are called last.

One method type, ***:default***, has a standard meaning independent of the style of method combination, and can be used with any style.

Here are the standardly defined method combination styles:

:daemon *[Method-combination type]*

■ The default style of method combination. All the ***:before*** methods are called, then the primary (untyped) method for the outermost flavor that has one is called, then all the ***:after*** methods are called. The value returned is the value of the primary method.

:daemon-with-or *[Method-combination type]*

■ Like the ***:daemon*** method combination style, except that the primary method is wrapped in an ***:or*** special form with all ***:or*** methods. Multiple values can be returned from the primary method, but not from the ***:or*** methods (as in the **or** special form). This produces combined methods like the following:

```
(progn
  (foo-before-method)
  (multiple-value-prog1
    (or (foo-or-method)
        (foo-primary-method))
    (foo-after-method)))
```

This is useful primarily for flavors in which a mixin introduces an alternative to the primary method. Each ***:or*** method gets a chance to run before the primary method and to decide whether the primary method should be run or not; if any ***:or*** method returns a non-*nil* value, the primary method is not run (nor are the rest of the ***:or*** methods). Note that the ordering of the combination of the ***:or*** methods is controlled by the order keyword in the ***:method-combination*** option.

:daemon-with-and *[Method-combination type]*

■ Like ***:daemon-with-or*** except that it combines ***:and*** methods in an **and** special form. The primary method is run only if all of the ***:and*** methods return non-*nil* values.

:daemon-with-override *[Method-combination type]*

■ Like the ***:daemon*** method combination style, except an **or** special form is wrapped around the entire combined method with all ***:override*** typed methods before the combined method. This differs from ***:daemon-with-or*** in that the ***:before*** and ***:after*** daemons are run only if *none* of the ***:override*** methods returns non-*nil*. The combined method looks something like this:

```
(or (foo-override-method)
    (progn
      (foo-before-method)
      (multiple-value-prog1
        (foo-primary-method)
        (foo-after-method))))
```

:progn *[Method-combination type]*

■ Calls all the methods inside a **progn** special form. Only untyped and ***:progn*** methods are allowed. The combined method calls all the ***:progn*** methods and then all the untyped methods. The result of the combined method is whatever the last of the methods returns.

:or *[Method-combination type]*

■ Calls all the methods inside an **or** special form. This means that each of the methods is called in turn. Only untyped methods and ***:or*** methods are allowed; the ***:or*** methods are called first. If a method returns a non-*nil* value, that value is returned and none of the rest of the methods are called; otherwise, the next method is called. In other words, each method is given a chance to handle the message; if it doesn't want to handle the message, it can return *nil*, and the next method gets a chance to try.

:and *[Method-combination type]*

■ Calls all the methods inside an **and** special form. Only untyped methods and ***:and*** methods are allowed. The basic idea is much like ***:or***, see above.

:append *[Method-combination type]*

■ Calls all the methods and appends the values together. Only untyped methods and ***:append*** methods are allowed; the ***:append*** methods are called first.

:nconc *[Method-combination type]*

■ Calls all the methods and **nconc**s the values together. Only untyped methods and ***:nconc*** methods are allowed, etc.

:list *[Method-combination type]*

■ Calls all the methods and returns a list of their returned values. Only untyped methods and ***:list*** methods are allowed, etc.

:inverse-list *[Method-combination type]*

■ Calls each method with one argument; these arguments are successive elements of the list that is the sole argument to the operation. Returns no particular value. Only untyped methods and ***:inverse-list*** methods are allowed, etc.

□ If the result of a ***:list***-combined operation is sent back with an ***:inverse-list***-combined operation, with the same ordering and with corresponding method definitions, each component flavor receives the value that came from that flavor.

:pass-on *[Method-combination type]*

■ NOTE: ***:pass-on*** method combination is not yet implemented in Tek COMMON LISP.

■ Calls each method on the values returned by the preceding one. The values returned by the combined method are those of the outermost call. The format of the declaration in the **defflavor** is:

```
(:method-combination (:pass-on (ordering .
arglist)
'operation-names))
```

where ordering is ***:base-flavor-first*** or ***:base-flavor-last***. arglist may include the ***&aux*** and ***&optional*** keywords.

□ Only untyped methods and ***:pass-on*** methods are allowed. The ***:pass-on*** methods are called first.

:case *[Method-combination type]*

■ With ***:case*** method combination, the combined method automatically does a **caseq** dispatch on the first argument of the operation, known as the *suboperation.* Methods of type ***:case*** can be used, and each one specifies one suboperation that it applies to. If no ***:case*** method matches the suboperation, the primary method, if any, is called.

```
(defflavor foo (a b) ()
  (:method-combination (:case :base-flavor-last
:win)))

(defmethod (foo :case :win :a) ()
  ;; This method handles nd a-foo :win :a):
  a)

(defmethod (foo :case :win :a*b) ()
  ;; This method handles nd a-foo :win :a*b):
  (* a b))

(defmethod (foo :win) (suboperation)
  ;; This method handles nd a-foo :win
:something-else):
  (list 'something-random suboperation))
```

:case methods are unusual in that one flavor can have many ***:case*** methods for the same operation, as long as they are for different suboperations.

☐ The suboperations ***:which-operations***, ***:operation-handled-p***, ***:send-if-handles*** and ***:get-handler-for*** are all handled automatically based on the collection of ***:case*** methods that are present.

■ NOTE: ***:send-if-handles*** and ***:get-handler-for*** are presently unimplemented in Tek COMMON LISP.

☐ Methods of type ***:or*** are also allowed. They are called just before the primary method, and if one of them returns a non-*nil* value, that is the value of the operation, and no more methods are called.

Here is a list of all the method types recognized by the standard styles of method combination:

no method type *[Method type]*

■ If no type is given to **defmethod**, a primary method is created. This is the most common type of method.

:before *[Method type]*
:after *[Method type]*

■ These are used for the before-daemon and after-daemon methods used by ***:daemon*** method combination.

:default *[Method type]*

■ If there are no untyped methods among any of the flavors being combined, then the ***:default*** methods (if any) are treated as if they were untyped. If there are any untyped methods, the ***:default*** methods are ignored.

□ Typically a base-flavor defines some default methods for certain of the operations understood by its family. When using the default kind of method combination these default methods are suppressed if another component provides a primary method.

:or *[Method type]*
:and *[Method type]*

■ These are used for ***:daemon-with-or*** and ***:daemon-with-and*** method combination. The ***:or*** methods are wrapped in an ***or***, or the ***:and*** methods are wrapped in an **and**, together with the primary method, between the ***:before*** and ***:after*** methods.

:override *[Method type]*

■ Allows the features of ***:or*** method combination to be used together with daemons. If you specify ***:daemon-with-override*** method combination, you may use ***:override*** methods. The ***:override*** methods are executed first, until one of them returns non-*nil*. If this happens, that method's value(s) are returned and no more methods are used. If all the ***:override*** methods return *nil*, the ***:before***, primary and ***:after*** methods are executed as usual.

□ In typical usages of this feature, the ***:override*** method usually returns *nil* and does nothing, but in exceptional circumstances it takes over the handling of the operation.

:or *[Method type]*
:and *[Method type]*
:progn *[Method type]*
:list *[Method type]*
:inverse-list *[Method type]*
:pass-on *[Method type]*
:append *[Method type]*
:nconc *[Method type]*

■ Each of these methods types is allowed in the method combination style of the same name. In those method combination styles, these typed methods work just like untyped ones, but all the typed methods are called before all the untyped ones. These method types can be used with any method combination style; they have standard meanings independent of the method combination style being used.

:wrapper *[Method type]*

■ This is used internally by **defwrapper**.

:combined *[Method type]*

■ This is used internally for automatically-generated *combined* methods.

The most common form of combination is ***:daemon***. One thing may not be clear: when do you use a ***:before*** daemon and when do you use an ***:after*** daemon? In some cases the primary method performs a clearly-defined action and the choice is obvious: ***:before :launch-rocket*** puts in the fuel, and ***:after :launch-rocket*** turns on the radar tracking.

In other cases the choice can be less obvious. Consider the ***:init*** message, which is sent to a newly-created object. To decide what kind of daemon to use, we observe the order in which daemon methods are called. First the ***:before*** daemon of the instantiated flavor is called, then ***:before*** daemons of successively more basic flavors are called, and finally the ***:before*** daemon (if any) of the base flavor is called. Then the primary method is called. After that, the ***:after*** daemon for the base flavor is called, followed by the ***:after*** daemons at successively less basic flavors.

Now, if there is no interaction among all these methods, if their actions are completely independent, then it doesn't matter whether you use a ***:before*** daemon or an ***:after*** daemon. There is a difference if there is some interaction. The interaction we are talking about is usually done through instance variables; in general, instance variables are how the methods of different component flavors

communicate with each other. In the case of the ***:init*** operation, the init-plist can be used as well. The important thing to remember is that no method knows beforehand which other flavors have been mixed in to form this flavor; a method cannot make any assumptions about how this flavor has been combined, and in what order the various components are mixed.

This means that when a ***:before*** daemon has run, it must assume that none of the methods for this operation have run yet. But the ***:after*** daemon knows that the ***:before*** daemon for each of the other flavors has run. So if one flavor wants to convey information to the other, the first one should *transmit* the information in a ***:before*** daemon, and the second one should *receive* it in an ***:after*** daemon. So while the ***:before*** daemons are run, information is *transmitted*; that is, instance variables get set up. Then, when the ***:after*** daemons are run, they can look at the instance variables and act on their values.

In the case of the ***:init*** method, the ***:before*** daemons typically set up instance variables of the object based on the init-plist, while the ***:after*** daemons actually do things, relying on the fact that all of the instance variables have been initialized by the time they are called.

The problems become most difficult when you are creating a network of instances of various flavors that are supposed to point to each other. For example, suppose you have flavors for *buffers* and *streams*, and each buffer should be accompanied by a stream. If you create the stream in the ***:before :init*** method for buffers, you can inform the stream of its corresponding buffer with an init keyword, but the stream may try sending messages back to the buffer, which is not yet ready to be used. If you create the stream in the ***:after :init*** method for buffers, there will be no problem with stream creation, but some other ***:after :init*** methods of other mixins may have run and made the assumption that there is to be no stream. The only way to guarantee success is to create the stream in a ***:before*** method and inform it of its associated buffer by sending it a message from the buffer's ***:after :init*** method. This scheme–creating associated objects in ***:before*** methods but linking them up in ***:after*** methods–often avoids problems, because all the various associated objects used by various mixins at least exist when it is time to make other objects point to them.

Since flavors are not hierarchically organized, the notion of levels of abstraction is not rigidly applicable. However, it remains a useful way of thinking about systems.

6.13 Implementing flavors

An object that is an instance of a flavor is implemented as a hidden data type similar to a simple vector. The zeroth slot points to a *flavor descriptor*, and successive slots of the vector store the instance variables. Sometimes, for debugging, it is useful to know that svref is legal on an instance. However, it is of course a violation of the implicit contract with a flavor to use this fact in real code.

A *flavor descriptor* is a defstruct of type flavors::flavor. It is also stored on the flavors::flavor property of the flavor name. It contains, among other things, the name of the flavor, the size of an instance, the table of methods for handling operations, and information for accessing the instance variables. The function (describe-flavor flavor-name) will print much of this information in readable format. **defflavor** creates a *flavor-descriptor* for each flavor and links them together according to the dependency relationships between flavors. Much of the information stored there, of course, is not computed until flavor-combination time.

A message is sent to an instance simply by calling it as a function, with the first argument being the operation. The evaluator looks up the operation in the dispatch hashtable stored in the flavor descriptor for that flavor and obtains a handler function and a mapping table. It then binds self to the object, si::self-mapping-table to the mapping table. Finally, the handler function is called. If there is only one method to be invoked, the handler function is that method; otherwise it is an automatically-generated function, called the combined method, which calls the component methods appropriately. If there are wrappers, they are incorporated into the combined method.

The code body of each method function knows only about the instance variables declared for its flavor, and this set of instance variables is known when the defining **defmethod** is evaluated. However, the location of these instance variables within an instance of an arbitrary flavor containing that flavor is not known until flavor-combination time. The mapping table is used by a method to map the set of instance variables it knows about into slot offsets within self. If all the component methods invoked by the combined method derive from a single flavor, the mapping table obtained from the method dispatch hashtable is a simple vector of slot numbers. If methods from more than one component flavor are invoked from the combined method, then the mapping table is an *alist* mapping each component flavor to its appropriate component mapping table, and the combined method takes care of binding si:self-mapping-table appropriately before calling each component.

For both interpreted and compiled methods in Tek COMMON LISP all instance variables are lexical scoped within the body of the method. (This is different from the FRANZ LISP implementation, in which the interpreter cannot implement lexical scoping.)

6.13.1 Order of definition

There is a certain amount of freedom to the order in which you do **defflavors**, **defmethods**, and **defwrappers**. This freedom is designed to make it easy to load programs containing complex flavor structures without having to do things in a certain order. It is considered important that not all the methods for a flavor need be defined in the same file. Thus the partitioning of a program into files can be along modular lines.

The rules for the order of definition are as follows.

Before a method can be defined (with **defmethod** or **defwrapper**) its flavor must have been defined (with **defflavor**). This makes sense because the system has to have a place to remember the method, and because it has to know the instance-variables of the flavor if the method is to be compiled.

When a flavor is defined (with **defflavor**) it is not necessary that all of its component flavors be defined already. This is to allow **defflavors** to be spread between files according to the modularity of a program, and to provide for mutually-dependent flavors. Methods can be defined for a flavor some of whose component flavors are not yet defined; however, compilation of a method which refers to instance variables inherited from a flavor not yet defined, and not mentioned in a **:required-instance-variable** clause, will produce a compiler warning that the variable was declared special (because the system did not realize it was an instance variable). If this happens, you should fix the problem and recompile. It may be sufficient just to change the order in which the flavors are defined, but considerations of modularity, clarity, and self documentation make it far preferable to insert **:required-instance-variable** clauses.

The methods automatically generated by the ***:gettable-instance-variables***, ***:settable-instance-variables***, and ***:outside-accessible-instance-variables*** **defflavor** options are generated at the time the **defflavor** is done.

The first time a flavor is instantiated, or when **compile-flavor-methods** is done, the system looks through all of the component flavors and gathers various information. At this point an error is signaled if not all of the components have been **defflavored**. This is also the time at which certain other errors are detected, such as the lack of a required instance-variable (see the ***:required-instance-variables*** option to **defflavor**). The ordered set of instance variables is determined and their slots assigned within an instance. The combined methods are generated unless they already exist and are correct. The flavor system tries very hard never to **redefun** a combined method unless its contents actually must change.

After a flavor has been instantiated, it is possible to make changes to it. Such changes affect all existing instances if possible. This is described more fully immediately below.

6.13.2 Changing a flavor

You can change anything about a flavor at any time. You can change the flavor's general attributes by doing another **defflavor** with the same name. You can add or modify methods by doing **defmethod**s. If you do a **defmethod** with the same flavor-name, operation (and suboperation if any), and (optional) method-type as an existing method, that method is replaced by the new definition.

These changes always propagate to all flavors that depend upon the changed flavor. Normally the system propagates the changes to all existing instances of the changed flavor and its dependent flavors. However, this is not possible when the flavor has been changed in such a way that the old instances would not work properly with the new flavor. This happens if you change the number of instance variables, which changes the size of an instance. It also happens if you change the order of the instance variables (and hence the storage layout of an instance), or if you change the component flavors (which can change several subtle aspects of an instance). The system does not keep a list of all the instances of each flavor, so it cannot find the instances and modify them to conform to the new flavor definition. Instead it gives you a warning message to the effect that the flavor was changed incompatibly and the old instances will not get the new version. The system leaves the old flavor data-structure intact (the old instances continue to point at it) and makes a new one to contain the new version of the flavor. If a less drastic change is made, the system modifies the original flavor data-structure, thus affecting the old instances that point at it. However, if you redefine methods in such a way that they only work for the new version of the flavor, then trying to use those methods with the old instances won't work.

6.14 Property list operations

It is often useful to associate a property list with an abstract object, for the same reasons that it is useful to have a property list associated with a symbol. This section describes a mixin flavor, si:property-list-mixin, that can be used as a component of any new flavor in order to provide that new flavor with a property list. For more details and examples, see the general discussion of property lists. The usual property list functionalities (**get**, **putprop**, etc.) are obtained by sending the instance the corresponding message. The contents of the property list can be initialized by providing a ***:property-list*** init option on the init-plist given to **instantiate-flavor**.

si:property-list-mixin *[Flavor]*

■ This mixin flavor provides the basic operations on property lists.

:get property-name *[Message]*

■ Looks up the object's property-name property.

:getl property-name-list *[Message]*

■ Like the ***:get*** operation, except that the argument is a list of property names. The ***:getl*** operation searches down the property list until it finds a property whose property name is one of the elements of property-name-list. It returns the portion of the property list beginning with the first such property that it found. If it doesn't find any, it returns *nil*.

:putprop value property-name *[Message]*

■ Gives the object a property-name property of value.

:remprop property-name *[Message]*

■ Removes the object's property-name property, by splicing it out of the property list. It returns one of the cells spliced out, whose car is the former value of the property that was just removed. If there was no such property to begin with, the value is *nil*.

:push-property value ***property-name*** *[Message]*

■ The property-name property of the object should be a list (note that *nil* is a list and an absent property is *nil*). This operation sets the property-name property of the object to a list whose car is **value** and whose cdr is the former property-name property of the list. This is analogous to doing

```
(push value (get object property-name))
```

:property-list *[Message]*

■ RETURNS the list of alternating property names and values that implements the property list.

:set-property-list list *[Message]*

■ Sets the list of alternating property names and values that implements the property list to list.

6.15 Copying instances

There are no built-in techniques to copy instances because there are too many questions raised about what should be copied. These include:

- Do you or do you not send an ***:init*** message to the new instance? If you do, what init-plist options do you supply?
- If the instance has a property list, you should copy the property list (e.g. with **copylist**) so that **putprop** or **remprop** on one of the instances does not affect the properties of the other instance.
- If the instance is a port connected to a network, some of the instance variables represent an agent in another host elsewhere in the network. Should the copy talk to the same agent, or should a new agent be constructed for it?
- If the instance is a port connected to a file, should copying the stream make a copy of the file or should it make another stream open to the same file? Should the choice depend on whether the file is open for input or for output?

In general, you can see that in order to copy an instance one must understand a lot about the instance. One must know what the instance variables mean so that the values of the instance variables can be copied if necessary. One must understand what relations to the external environment the instance has so that new relations can be established for the new instance. One must even understand what the general concept 'copy' means in the context of this particular instance, and whether it means anything at all.

Copying is a generic operation, whose implementation for a particular instance depends on detailed knowledge relating to that instance. Modularity dictates that this knowledge be contained in the instance's flavor, not in a *general copying function*. Thus the way to copy an instance is to send it a message, as in (send object :copy). It is up to you to implement the operation in a suitable fashion, such as

```
(defflavor foo (a b c) ()
  (:initable-instance-variables a b))

(defmethod (foo :copy) ()
  (make-instance 'foo :a a :b b))
```

The flavor system chooses not to provide any default method for copying an instance, and does not even suggest a standard name for the copying message, because copying involves so many semantic issues.

If a flavor supports the ***:reconstruction-init-plist*** operation, a suitable copy can be made by invoking this operation and passing the result to **make-instance** along with the flavor name. This is because the definition of what the ***:reconstruction-init-plist*** operation should do requires it to address all the problems listed above. Implementing

this operation is up to you, and so is making sure that the flavor implements sufficient init keywords to transmit any information that is to be copied.

7 Profiling

7 Profiling

In order to speed up a large LISP program it is first necessary to determine the parts of the program where most of the time is being spent. Excl automatically counts the number of times each function is called. This information, along with the programmer's knowledge of which functions are large and/or time consuming, will pinpoint the parts of the program that should be optimized.

The code for recording function call counts is very small and fast (just one machine instruction per function call) thus for most applications it makes sense to permit function call counting to occur. However for certain time critical highly recursive functions, it may be desireable to instruct the compiler to omit the the function call counting code for certain functions. This can be done by setting the variable *compiler:*do-call-counts** to *nil* before compiling the function. The code which does the counting is compiled into the compiled functions and thus can be turned on or off on a function by function basis. The system code always does call counting.

function-call-report ***&optional*** number-to-report *[Function]*

■ For all interned symbols with compiled function definitions, gather information on the number of times they have been called since the last function call report, and clear the call counts at the same time. Then sort the functions in descending order of number of times called and print the function call information on the most called functions. The optional argument, number-to-report, determines how many functions are printed. number-to-report defaults to *50*. Functions which are anonymous (not associated with any interned symbol) will be omitted from this list.

function-call-list *[Function]*

■ This function returns a list of all the functions which were called at least once and their call counts, and it clears the call counts. The form of each list entry is (number-of-calls . function-name). The list is sorted in descending order of number-of-calls.

function-call-clear *[Function]*

■ Clear the call counts for all functions.

function-call-count function *[Function]*

■ Return the number of times function has been called. function can either be a compiled function object or a symbol with a compiled function object as its function definition.

get-and-zero-call-count function *[Function]*

■ Return the number of times the function has been called and zero the call count. function can either be a compiled function object or a symbol with a compiled function object as its function definition.

compiler:*do-call-counts* *[Variable]*

■ If non-*nil*, then when the compiler compiles a function it will add code to maintain a call count.

8 Extrinsic data and procedures

8 Extrinsic data and procedures

In a future release, Tek COMMON LISP will provide the abilities to access and modify extrinsic data and to use extrinsic procedures.

A Summary of symbols

Summary of symbols

Index

Index

Tektronix 4400 graphics library

Tektronix 4400 graphics library

1 Introduction

The LISP *tek4400-graphics* (nicknamed *gr*) package permits LISP users to directly call functions in the graphics library on the 4400 series machines. The LISP interface is nearly identical to the C interface. Please refer to the Graphics Library documentation in the Workstation Reference Manual for information on what each graphics routine does. Listed below are the LISP function names and the reference to the C graphics name in the *4400 Series C Reference*.

The tek4400-graphics module can be loaded automatically with COMMON LISP **require** function as in (require 'tek-graph). Once loaded, the graphics mode must be initialized by calling the function **initialize-tek-graphics** which sets the **screen** variable. **screen** is the *form* representing the display bitmap.

2 Data structures

2.1 Points

A *point* is a special data type which internally consists of two 16-bit signed integers.

make-point x y *[Function]*

- Returns a *point*. x and y are integers.

point-x p *[Function]*
point-y p *[Function]*

- Accesses *point* p's x and y coordinates respectively.
- ☐ To modify the x and y coordinates of a point p, use **setf** as in (setf (point-x p) x2), where x2 is an integer.

pointp p *[Function]*

■ Returns *t* if and only if p is a *point.*

2.2 Rectangles

A *rectangle* is a special data type internally represented as four 16-bit signed integers.

make-rect x y w h *[Function]*

■ Returns a *rectangle.* x, y, w, and h are each integers.

rect-x r *[Function]*
rect-y r *[Function]*
rect-w r *[Function]*
rect-h r *[Function]*

■ Accesses *rectangle* r's x, y, w, and h corrdinates respectively.
□ To modify these coordinates of rectangle r, use **setf** as in (setf (rect-x r) x2), where x2 is an integer.

rectp v *[Function]*

■ Returns *t* if and only if v is a *rectangle.*

2.3 Forms

A *form* is a bitmap. The contents of a *form* should only be manipulated with such functions as **bit-blt** or **paint-line**; the fields should never be modified in any other way by a user program! Doing so may cause the low level *bit-blt* primitive to overwrite adjacent LISP objects and cause LISP to crash. *Forms* created by the **form-create** function contain the bitmap in the *form* (thus unless you set **print-array** to *nil* or set **print-length** to a small value you will not want 'print' to ever print a *form* value). The *form* returned by **init-graphics** points to the screen *form* (which is not in the *form* itself but which is pointed to by the *form*). Each *form* begins with a 'magic' number to aid type checking.

form-create w h *[Function]*

■ Creates and returns a *form.* This function is described in the 'Functions' section below.
□ See the C function **FormCreate** in §5 of the *4400 Series C Reference.*

form-w f *[Function]*
form-h f *[Function]*

■ Accesses *form* f's w (width) and h (height) dimensions respectively.

formp v *[Function]*

■ Returns *t* if and only if v is a *form*.

2.4 Bbcoms

A *bbcom* is a *bit-blt* command vector. The user constructs the command vector and then passes it to such functions as **bit-blt** or **paint-line**. The contents of the *bbcom* may be modified, but only with legal values. Fields are:

srcform	a *form* or *nil*
destform	a *form*
srcpoint	a *point*
destrect	a *rect*
cliprect	a *rect*
halftoneform	a *form* or *nil*
rule	a *fixnum*

make-bbcom ***&key :srcform :destform :srcpoint :destrect :cliprect :halftoneform :rule*** *[Function]*

■ Returns a *bbcom*. The default values are

srcform	*nil*
destform	**screen**
halftoneform	*nil*
srcpoint	*(0,0)*
destrect	*(0,0,*screen-width*,*screen-height*)*
cliprect	*(0,0,*screen-width*,*screen-height*)*
rule	*bb-s* if srcform is non-*nil* or *bb-one* if srcform is *nil*.

screen names the *form* for the screen. **screen-width** and **screen-height** name the pixel width and height of the screen respectively.

☐ For each of the fields *X*, there is a **bbcom-*X*** function to access that field.
☐ To modify a field *X*, one can use **setf** for each *X* as in (setf (bbcom-halftoneform form) r2) where form is a *form*, and r2 is a bbcom rule.
☐ This function replaces the **BbcomDefault** function in the C graphics library.

bbcomp v *[Function]*

■ Returns *t* if and only if *v* is a *bbcom*.

2.5 Display states

A *display state* holds the complete state of the display (except the screen bitmap).

make-display-state *[Function]*

■ Returns a *display state*.

display-state-p v *[Function]*

■ Returns *t* if and only if v is a *display state*.

2.6 Menus

A menu is a structure containing forms, a *bbcom* and several other fields.

icon-menu-create icon-vector *[Function]*

■ Initializes and returns a *menu*. This function is described in the 'Functions' section below.
☐ See the C function **IconMenuCreate** in §5 of the *4400 Series C Reference*.

icon-menu-create-x icon-vector flag-vector previous *[Function]*

■ Initializes and returns a *menu*. This function is described in the 'Functions' section below.
☐ See the C function **IconMenuCreateX** in §5 of the *4400 Series C Reference*.

menu-create vector-of-strings *[Function]*

■ Initializes and returns a *menu*. This function is described in the 'Functions' section below.
☐ See the C function **MenuCreate** in §5 of the *4400 Series C Reference*.

menu-create-x vector-of-strings flag-vector previous font *[Function]*

■ Initializes and returns a *menu*. This function is described in the 'Functions' section below.
☐ See the C function **MenuCreateX** in §5 of the *4400 Series C Reference*.

menu-destroy menu *[Function]*

■ Deallocates a *menu*. This function is described in the 'Functions' section below.
☐ See the C function **MenuDestroy** in §5 of the *4400 Series C Reference*.

menu-left *[Constant]*
menu-noselect *[Constant]*
menu-right *[Constant]*

■ Constants for the flag vectors of *menu* structures.

bb-zero *[Constant]*
bb-s-and-d *[Constant]*
bb-s-and-not-d *[Constant]*
bb-s *[Constant]*
bb-not-s-and-d *[Constant]*
bb-d *[Constant]*
bb-s-xor-d *[Constant]*
bb-s-or-d *[Constant]*
bb-not-s-and-not-d *[Constant]*
bb-not-s-xor-d *[Constant]*
bb-not-d *[Constant]*
bb-s-or-not-d *[Constant]*
bb-not-s *[Constant]*
bb-d-or-not-s *[Constant]*
bb-not-s-or-not-d *[Constant]*
bb-one *[Constant]*

3 Rules for **bit-blt**

■ These LISP constants are the rules passed to **bit-blt** and correspond to the C constants defined in */lib/include/graphics.h*.

screen-height *[Variable]*
screen-width *[Variable]*

4 Variables

■ These variables contain the size in pixels, as height and width, of the screen bitmap.

view-height *[Variable]*
view-width *[Variable]*

■ These variables contain the size in pixels, as height and width, of the visible portion of the screen bitmap.

5 Functions

The documentation for these functions can be found in §5, 'Graphics Library Reference,' of the *4400 Series C Reference*. The order that the functions are listed below is the same order as they appear in that section of the manual, i.e. in alphabetical order. Note that some of the functions have been described above, and that some functions in the C library have no counterpart in LISP. Functions that have no counterpart are functions such as **BbcomDefault** and **FormCopy** that manifest the unique programming paradigm of C. In the case of **BbcomDefault**, the LISP counterpart is **make-bbcom**, which both allocates and initializes a *bbcom* object. In C a *bbcom* would be created by declaring a struct BBCOM structure, whereas in LISP a *bbcom* object must be allocated. In the case of **FormCopy**, the generic LISP function **copy-seq** subsumes the data-specific C function.

In general the LISP function's name is generated by a simple transformation of the C function name. The C function name is divided just before each embedded capital letter, the letter is converted to lower case, and a hyphen is inserted at each such division. For example, the C library function **ClearScreen** becomes the LISP function **clear-screen**. In some cases the LISP function expands abbreviated components in C function names, e.g. the C function **GetCPosition** becomes the LISP function **get-cursor-position**.

Certain conventions are observed in describing the arguments to these functions. Arguments ending with *"p"* are predicates. The value *nil* means 'false' and anything else means 'true'. Some of the functions that return structures take optional arguments (e.g. **get-viewport**). If the argument is passed to the function, then the result value will be stored in that argument, otherwise the function will allocate a new structure and return it.

N.B. All functions check for errors from the library and signal an error if an error is detected. Thus functions don't return if there was an error. If there was no error, the functions return the value returned by the C library function, converted to an appropriate LISP data type.

bit-blt bbcom *[Function]*

■ Performs the **bit-blt** command described in the bbcom record.
☐ See the C function **BitBlt** in §5 of the *4400 Series C Reference.*

char-draw char point *[Function]*

■ Draws the *character* char at *point* point.
☐ See the C function **CharDraw** in §5 of the *4400 Series C Reference.*

char-draw-raw-x char point bbcom font *[Function]*

■ Draws the *character* char in *font* font at *point* point using the parameters of the *bbcom.*
☐ See the C function **CharDrawRawX** in §5 of the *4400 Series C Reference.*

char-draw-x char point bbcom font *[Function]*

■ Draws the *character* char in *font* font at *point* point using the parameters of the *bbcom.* The value of point is updated to location at the end of the character.
☐ See the C function **CharDrawX** in §5 of the *4400 Series C Reference.*

char-width char font *[Function]*

■ Returns the width in pixels required to draw *character* char in *font* font.
☐ See the C function **CharFont** in §5 of the *4400 Series C Reference.*

circle-draw center radius *[Function]*

■ Draws a circle centered at *point* center of the specified *fixnum* radius.
☐ See the C function **CircleDraw** in §5 of the *4400 Series C Reference.*

circle-draw-x center radius width bbcom *[Function]*

■ Draws a circle centered at *point* center of the specified *fixnum* radius using a line of *fixnum* width onto the form specified by bbcom.
☐ See the C function **CircleDrawX** in §5 of the *4400 Series C Reference.*

clear-screen *[Function]*

■ Clear the screen.
☐ See the C function **ClearScreen** in §5 of the *4400 Series C Reference*.

cursor-track trackp *[Function]*

■ Force cursor to track the mouse if trackp is true.
☐ See the C function **CursorTrack** in §5 of the *4400 Series C Reference*.

cursor-visible visiblep *[Function]*

■ Make cursor visible or invisible based on visiblep
☐ See the C function **CursorVisible** in §5 of the *4400 Series C Reference*.

display-visible visiblep *[Function]*

■ Make the display visible or invisible based on visiblep.
☐ See the C function **DisplayVisible** in §5 of the *4400 Series C Reference*.

event-clear-alarm *[Function]*

■ Clears any pending alarms that the process has requested.
☐ See the C function **EClearAlarm** in §5 of the *4400 Series C Reference*.

event-disable *[Function]*

■ Disables event processing.
☐ See the C function **EventDisable** in §5 of the *4400 Series C Reference*.

event-enable *[Function]*

■ Enables event processing.
☐ See the C function **EventEnable** in §5 of the *4400 Series C Reference*.

event-get-count *[Function]*

■ Returns the number of event vvalues in the event buffer waiting to be processed.
☐ See the C function **EGetCount** in §5 of the *4400 Series C Reference*.

event-get-new-count *[Function]*

■ Returns the number of event values in the event buffer which have occurred since the previous call to this function.
☐ See the C function **EGetNewCount** in §5 of the *4400 Series C Reference.*

event-get-next *[Function]*

■ Returns two values: an event-type code and an event value. The event-type codes are shown below.

0	delta time
1	mouse *x* location
2	mouse *y* location
3	key or button pressed
4	key or button released
5	absolute time

☐ Whenever the keyboard or mouse changes state, a time event is generated (either a type 0 or type 5 event) that reports the time of the event. This is accompanied by an event value that specifies the actual change that occurred.
☐ See the C function **EGetNext** in §5 of the *4400 Series C Reference.*

event-get-time *[Function]*

■ Returns the time, in milliseconds, since the system was powered up.
☐ See the C function **EGetTime** in §5 of the *4400 Series C Reference.*

event-set-mouse-interval interval *[Function]*

■ Specifies how frequently mouse motion events are to be created if the mouse is continuously moving. interval is a *fixnum.*
☐ See the C function **ESetMInterval** in §5 of the *4400 Series C Reference.*

event-set-alarm time *[Function]*

■ Requests a signal when the specified time, in milliseconds, is reached.
☐ See the C function **ESetAlarm** in §5 of the *4400 Series C Reference.*

event-set-signal *[Function]*

■ Requests the event manager to signal the current process when events occur.
☐ See the C function **ESetSignal** in §5 of the *4400 Series C Reference.*

exit-graphics *[Function]*

■ Exit graphics mode.
☐ See the C function **ExitGraphics** in §5 of the *4400 Series C Reference.*

font-close font *[Function]*

■ Releases storage used for the specified *font* font.
☐ See the C function **FontClose** in §5 of the *4400 Series C Reference.*

font-open font-file *[Function]*

■ Initializes a font from the font file font-file.
☐ See the C function **FontOpen** in §5 of the *4400 Series C Reference.*

form-draw form *[Function]*

■ Displays the *form* form.

form-create w h *[Function]*

■ Creates and returns a *form* with width w and height h.
☐ See the C function **FormCreate** in §5 of the *4400 Series C Reference.*

form-get-point form ***&optional*** point *[Function]*

■ Returns the value of a particular *point* (default *(0, 0)*) in a form.
☐ See the C function **FormGetPoint** in §5 of the *4400 Series C Reference.*

form-read file-name *[Function]*

■ Reads a file in Smalltalk 'form' format from disk and returns a form object initialized from the file.
☐ See the C function **FormRead** in §5 of the *4400 Series C Reference.*

form-set-point form point value *[Function]*

■ Sets the value of a single *point* in form to value.
□ See the C function **FormSetPoint** in §5 of the *4400 Series C Reference*.

form-write form file-name *[Function]*

■ Writes the *form* form to the file file-name in Smalltalk format.
□ See the C function **FormWrite** in §5 of the *4400 Series C Reference*.

get-buttons *[Function]*

■ This returns an integer whose lower three bits are the mouse button values (1 ≡ down).
□ See the C function **GetButtons** in §5 of the *4400 Series C Reference*.

get-cursor ***&optional*** form *[Function]*

■ Returns the cursor image bitmap. The image will be stored in form if provided.
□ See the C function **GetCursor** in §5 of the *4400 Series C Reference*.

get-cursor-position ***&optional*** point *[Function]*

■ Get the position where the cursor is currently displayed.
□ See the C function **GetCPosition** in §5 of the *4400 Series C Reference*.

get-machine-type *[Function]*

■ Returns the 4400-series model number as set at machine initialization time or by the function **set-machine-type**.
□ See the C function **GetMachineType** in §5 of the *4400 Series C Reference*.

get-real-machine-type *[Function]*

■ Returns the 4400-series model number stored in internal ROM.
□ See the C function **GetRealMachineType** in §5 of the *4400 Series C Reference*.

get-mouse-bounds ***&optional*** point1 point2 *[Function]*

■ Get the limits on mouse motion. The *points* will be stored in point1 and point2 if provided.
☐ See the C function **GetMBounds** in §5 of the *4400 Series C Reference.*

get-mouse-position ***&optional*** point *[Function]*

■ Get the position where the mouse is currently pointing. The point will be stored in point if provided.
☐ See the C function **GetMPosition** in §5 of the *4400 Series C Reference.*

get-term-em-rc *[Function]*

■ Returns two *fixnum* values, the number of rows and columns, of the terminal emulator.
☐ See the C function **GetTermEmRC** in §5 of the *4400 Series C Reference.*

get-viewport ***&optional*** point *[Function]*

■ Get the position which the panning hardware is displaying as the upper left-hand corner of the display.
☐ See the C function **GetViewport** in §5 of the *4400 Series C Reference.*

icon-menu-create icon-vector *[Function]*

■ Initializes and returns a *menu.* The argument icon-vector is a vector of pointers to *forms* or *nil.*
☐ See the C function **IconMenuCreate** in §5 of the *4400 Series C Reference.*

icon-menu-create-x icon-vector flag-vector previous *[Function]*

■ Initializes and returns a *menu.* The argument icon-vector is a vector of pointers to *forms* or *nil.* The argument flag-vector must be an array of *(signed-byte 32)* elements of the same length as icon-vector. The *fixnum* parameter previous specifies the initial mouse position.
☐ See the C function **IconMenuCreateX** in §5 of the *4400 Series C Reference.*

init-graphics ***&optional*** set-full-graphics-mode-p *[Function]*

■ Initialize display for graphics.
☐ See the C function **InitGraphics** in §5 of the *4400 Series C Reference*. It is recommended to use **initialize-tek-graphics.**

initialize-tek-graphics *[Function]*

■ This function sets the **screen** variable and must be called before doing any graphics operations.
☐ This function is equivalent to (setq *screen* (init-graphics nil)).

line-draw point-1 point-2 *[Function]*

■ Draws a one-pixel wide line between the *points* point-1 and point-2. Both endpoints are drawn.
☐ See the C function **LineDraw** in §5 of the *4400 Series C Reference.*

line-draw-x point-1 point-2 width draw-last-p bbcom *[Function]*

■ Draws a line of width width pixels between the *points* point-1 and point-2 onto the *form* specified by *bbcom.* Both endpoints are drawn unless draw-last-p is *nil* and width is *1.*
☐ See the C function **LineDrawX** in §5 of the *4400 Series C Reference.*

menu-create vector-of-strings *[Function]*

■ Initializes and returns a *menu.* The argument vector-of-strings must be of type (simple-array (simple-string *) (*)).
☐ See the C function **MenuCreate** in §5 of the *4400 Series C Reference.*

menu-create-x vector-of-strings flag-vector previous font *[Function]*

■ Initializes and returns a *menu.* The argument vector-of-strings must be of type (simple-array (simple-string *) (*)). The argument flag-vector must be of type (array (signed-byte 32)) and of the same length as vector-of-strings. The *fixnum* argument previous specifies the initial mouse position. Menu items are displayed in *font* font.
☐ See the C function **MenuCreateX** in §5 of the *4400 Series C Reference.*

menu-destroy menu *[Function]*

■ The *menu* menu is deallocated.
☐ See the C function **MenuDestroy** in §5 of the *4400 Series C Reference*.

menu-select menu *[Function]*

■ Opens the specified *menu* and waits for a mouse click or release.
☐ See the C function **MenuSelect** in §5 of the *4400 Series C Reference*.

paint-line bbcom point *[Function]*

■ Paints a line on the display.
☐ See the C function **PaintLine** in §5 of the *4400 Series C Reference*.

pan-cursor-enable enablep *[Function]*

■ Enable screen panning using the cursor if enablep is true.
☐ See the C function **PanCursorEnable** in §5 of the *4400 Series C Reference*.

pan-disk-enable enablep *[Function]*

■ Enable screen panning using the joydisk if enablep is true.
☐ See the C function **PanDiskEnable** in §5 of the *4400 Series C Reference*.

point-distance point-1 point-2 *[Function]*

■ Returns the distance between the two *points*.
☐ See the C function **PointDistance** in §5 of the *4400 Series C Reference*.

point-from-user ***point*** *[Function]*

■ Returns a point selected by the user.
☐ See the C function **PointFromUser** in §5 of the *4400 Series C Reference*.

point-max point-1 point-2 *[Function]*

■ Returns the lower right corner of the rectangle defined the the two *points*.
☐ See the C function **PointMax** in §5 of the *4400 Series C Reference*.

point-midpoint point-1 point-2 *[Function]*

■ Returns the midpoint of the line defined by the two *points*.
☐ See the C function **PointMidpoint** in §5 of the *4400 Series C Reference*.

point-min point-1 point-2 *[Function]*

■ Returns the upper left corner of the rectangle defined by the two *points*.
☐ See the C function **PointMin** in §5 of the *4400 Series C Reference*.

point-to-row-column point *[Function]*

■ Converts a screen coordinate to the row, column indices which define the terminal emulator character cell which contains that point. Returns two values: the row and column.
☐ See the C function **PointToRC** in §5 of the *4400 Series C Reference*.

points-to-rect point-1 point-2 *[Function]*

■ Returns the minimum rectangle that contains both *points*.
☐ See the C function **PointsToRect** in §5 of the *4400 Series C Reference*.

polygon-draw point-vector *[Function]*

■ Draws a filled-in polygon defined by the points in the *simple-vector* point-vector.
☐ See the C function **PolygonDraw** in §5 of the *4400 Series C Reference*.

polygon-draw-x point-vector bbcom *[Function]*

■ Draws a filled-in polygon defined by the points in the *simple-vector* point-vector onto the destination form specified by bbcom.
☐ See the C function **PolygonDrawX** in §5 of the *4400 Series C Reference*.

polyline-draw point-vector *[Function]*

■ Draws a series of line segments connecting the points of the *simple-vector* point-vector using the *bb-s-or-d* combination rule. The line segments are one-pixel wide.
☐ See the C function **PolyLineDraw** in §5 of the *4400 Series C Reference*.

polyline-draw-x point-vector width closed bbcom *[Function]*

■ Draws a series of line segments connecting the points of the *simple-vector* point-vector, each line of width width pixels onto the form specified by bbcom. If the *fixnum* closed is not zero, then a closing line segment is drawn from the last to the first point in the vector. The last endpoint is not drawn.
☐ See the C function **PolyLineDrawX** in §5 of the *4400 Series C Reference.*

protect-cursor rect1 ***&optional*** rect2 *[Function]*

■ Tell the operating system to respond by removing the cursor from the screen if it is in either rect1 or (optionally) rect2.
☐ See the C function **ProtectCursor** in §5 of the *4400 Series C Reference.*

rect-areas-differing rectangle-1 rectangle-2 *[Function]*

■ Returns the regions of rectangle-1 that are outside of rectangle-2, and the regions of rectangle-2 that are outside of rectangle-1.
☐ See the C function **RectAreasDiffering** in §5 of the *4400 Series C Reference.*

rect-areas-outside rectangle-1 rectangle-2 *[Function]*

■ Returns the regions of rectangle-1 that are outside of rectangle-2.
☐ See the C function **RectAreasOutside** in §5 of the *4400 Series C Reference.*

rect-box-draw rectangle width *[Function]*

■ Draws a box of *fixnum* width pixels around rectangle using the *bbSorD* combination rule.
☐ See the C function **RectBoxDraw** in §5 of the *4400 Series C Reference.*

rect-box-draw-x rectangle width bbcom *[Function]*

■ Draws a box of *fixnum* width pixels around rectangle onto the form specified by bbcom.
☐ See the C function **RectBoxDrawX** in §5 of the *4400 Series C Reference.*

rect-contains-point rectangle point *[Function]*

■ Returns *nil* if rectangle does not contain point, otherwise it returns *t*.
☐ See the C function **RectContainsPoint** in §5 of the *4400 Series C Reference*.

rect-contains-rect rectangle-1 rectangle-2 *[Function]*

■ Returns *nil* unless rectangle-1 contains rectangle-2, in which case it returns *t*.
☐ See the C function **RectContainsRect** in §5 of the *4400 Series C Reference*.

rect-draw rectangle *[Function]*

■ Draws a solid rectangle using the *bbS* combination rule.
☐ See the C function **RectDraw** in §5 of the *4400 Series C Reference*.

rect-draw-x rectangle bbcom *[Function]*

■ Draws a solid rectangle onto the form specified by bbcom.
☐ See the C function **RectDrawX** in §5 of the *4400 Series C Reference*.

rect-from-user ***&optional*** rectangle *[Function]*

■ The region selected by the user is returned.
☐ See the C function **RectFromUser** in §5 of the *4400 Series C Reference*.

rect-from-user-x minimum-size form ***&optional*** rectangle *[Function]*

■ The region selected by the user is returned. The minimum-size argument specifies the minimum size of the region. The form argument specifies a half-tone form to highlight the selected region.
☐ See the C function **RectFromUserX** in §5 of the *4400 Series C Reference*.

rect-intersect rectangle-1 rectangle-2 ***&optional*** rectangle-3 *[Function]*

■ Returns (in rectangle-3 if given) the intersection of rectangle-1 and rectangle-2.

☐ See the C function **RectIntersect** in §5 of the *4400 Series C Reference.*

rect-intersects rectangle-1 rectangle-2 *[Function]*

■ Returns *t* if the rectangles intersect, otherwise *nil.*
☐ See the C function **RectIntersects** in §5 of the *4400 Series C Reference.*

rect-merge rectangle-1 rectangle-2 ***&optional*** rectangle-3 *[Function]*

■ Returns (in rectangle-3 if given) the minimum rectangle that contains both rectangle-1 and rectangle-2.
☐ See the C function **RectMerge** in §5 of the *4400 Series C Reference.*

release-cursor *[Function]*

■ Tell the operating system to restore the cursor if it was removed due to a call to **protect-cursor.**
☐ See the C function **ReleaseCursor** in §5 of the *4400 Series C Reference.*

restore-display-state display-state *[Function]*

■ Re-establish the state defined by display-state.
☐ See the C function **RestoreDisplayState** in §5 of the *4400 Series C Reference.*

row-column-to-rect row column ***&optional*** rect *[Function]*

■ Returns rectange which describes the terminal emulator character cell given by row and column. The rectangle will be returned in rect if provided.
☐ See the C function **RCToRect** in §5 of the *4400 Series C Reference.*

save-display-state ***&optional*** display-state *[Function]*

■ Return the *display-state* of the current display state. The display-state will be stored in display-state if provided.
☐ See the C function **SaveDisplayState** in §5 of the *4400 Series C Reference.*

screen-saver-enable enablep *[Function]*

■ Enable the screen saver timeout, which causes the screen to be blanked after 10 minutes of keyboard or mouse inactivity.
☐ See the C function **ScreenSaverEnable** in §5 of the *4400 Series C Reference.*

set-cursor form *[Function]*

■ Install a new cursor. form must be a 16x16 bit *form.*
☐ See the C function **SetCursor** in §5 of the *4400 Series C Reference.*

set-cursor-position point *[Function]*

■ Display the cursor at the specified position.
☐ See the C function **SetCPosition** in §5 of the *4400 Series C Reference.*

set-keyboard-code val *[Function]*

■ Tells the keyboard to output either event codes, if val is *0*, or ANSI character strings, if val is *1*.
☐ See the C function **SetKBCode** in §5 of the *4400 Series C Reference.*

set-machine-type value *[Function]*

■ Sets the machine type to the *fixnum* value.
☐ See the C function **SetMachineType** in §5 of the *4400 Series C Reference.*

set-mouse-bounds point1 point2 *[Function]*

■ Set the limits on mouse motion to be the rectangle defined by the upper left *point* point1 and the lower right *point* point2.
☐ See the C function **SetMBounds** in §5 of the *4400 Series C Reference.*

set-mouse-position point *[Function]*

■ Position the mouse at the specified position.
☐ See the C function **SetMPosition** in §5 of the *4400 Series C Reference.*

set-viewport point *[Function]*

■ Set the panning hardware to display the upper left-hand corner of the display at the specified point.
☐ See the C function **SetViewport** in §5 of the *4400 Series C Reference.*

string-draw string point *[Function]*

■ Draws the *simple-string* string using the default font starting at the specified point.
☐ See the C function **StringDraw** in §5 of the *4400 Series C Reference.*

string-draw-raw-x string point bbcom font *[Function]*

■ Draws the *simple-string* string using *font* font starting at the specified point onto the form specified by bbcom.
☐ See the C function **StringDrawRawX** in §5 of the *4400 Series C Reference.*

string-draw-x string point bbcom font *[Function]*

■ Draws the *simple-string* string using *font* font starting at the specified point onto the form specified by bbcom.
☐ See the C function **StringDrawX** in §5 of the *4400 Series C Reference.*

string-width string font *[Function]*

■ Returns the width in pixels of the *simple-string* string in *font* font.
☐ See the C function **StringWidth** in §5 of the *4400 Series C Reference.*

terminal-enable enablep *[Function]*

■ Enable the terminal emulator if enablep is true. The previous mode is returned (*t* for enabled, *nil* for disabled).
☐ See the C function **TerminalEnable** in §5 of the *4400 Series C Reference.*

video-normal normalp *[Function]*

■ Set display to white on black if normalp is *nil*, and to black on white otherwise.
☐ See the C function **VideoNormal** in §5 of the *4400 Series C Reference.*

6 Halftone forms

Tek COMMON LISP includes a function for creating halftone forms, and several variables that represent common halftone forms.

make-halftoneform ***&optional*** patternlist *[Function]*

■ Make a halftone which has the given pattern in it. patternlist is normally a list of sixteen 16-bit signed integers. If patternlist has fewer than sixteen integers, then the whole pattern is repeated as many times as is necessary to get sixteen integers.

black-halftone *[Variable]*
dark-grey-halftone *[Variable]*
grey-halftone *[Variable]*
light-grey-halftone *[Variable]*
very-light-grey-halftone *[Variable]*
*white-halftone** *[Variable]*

■ Various common halftone forms.

■

The *Tek COMMON LISP User Guide* was printed on an Apple Laserwriter laser printer driven by Adobe Systems' Postscript. The manual was typeset using the Unix device-independent **ditroff** program, with tables preprocessed by **dtbl** and equations by **deqn**. The Index, Contents, and Appendix A were generated automatically. The text is set in Times Roman and Helvetica. Examples are set in Courier.

■

Information Sheet

This sheet provides the postal and electronic mail addresses to which bugs, comments, suggestions, and other correspondence may be addressed. We look forward to hearing from you!

Bug reports

Franz Incorporated
Technical Support Group
1141 Harbor Bay Parkway
Alameda, California 94501

415 769 5656

...!ucbvax!franz!bugs
bugs%franz.uucp@kim.berkeley.edu

Subscription requests to the electronic forum

...!ucbvax!excl-forum-request
excl-forum-request@ucbvax.berkeley.edu

Contributions to the electronic forum

...!ucbvax!excl-forum
excl-forum@ucbvax.berkeley.edu

Mailing list additions

Franz Incorporated
Mailing Lists
1141 Harbor Bay Parkway
Alameda, California 94501

415 769 5656

...!ucbvax!franz!info
info%franz.uucp@kim.berkeley.edu

General correspondence

Franz Incorporated
1141 Harbor Bay Parkway
Alameda, California 94501

415 769 5656

...!ucbvax!franz!info
info%franz.uucp@kim.berkeley.edu

Customer Comment Sheet

We invite your comments and suggestions about this manual–how it can be improved and how it has been helpful. If you find any errors (typographical, of omission, of content, or otherwise), please let us know so that we may correct them in our next edition. This form is provided only for your convenience–we always welcome personal correspondence. Thank you for taking the time to help us improve the manual.

Your name _____

Company _____

Address _____

_____

City _____

State _____ Postal Code _____

Nation _____

Telephone _____

Operating system _____

Hardware _____

Extended COMMON LISP release _____

Errors, omissions, and inconsistencies:

Suggestions for Improving the manual:

Features of this manual you found helpful:

For further comments, please use the other side of this form. Thank you.

Release 1.0 notes for Tektronix

This chapter describes release **1.0** of Tek COMMON LISP for Tektronix workstations. The information provided here pertains specifically to this release–general information on the topics discussed here may be found in the *Tek COMMON LISP User Guide* and in *Common Lisp: The Language.*

1 Functionality

An extension to COMMON LISP, the foreign-function interface, is not part of this release. It is currently in development and will be incorporated into a future release.

It is not possible to compile functional objects (closures) created by interpretive evaluation of the **function** special form. For example, the functional value of symbol closure-sym below cannot be compiled in this release, and an error will be signalled by the **compile** function.

```
<cl> (setf (symbol-function 'closure-sym)
           (let ((local-var 0))
             (function
              (lambda (bound-var)
                (+ bound-var
                   local-var)))))
<cl> (compile 'closure-sym)
```

2 Arrays

The functions **bit, sbit, char** and **schar** are treated just like **aref** in that no effort is made to ensure that the first argument is of the correct type. It is an error, although it is not signalled, to provide a sequence argument of the wrong type.

3 Input/output

The variable **print-circle** is ignored by the printer in this release. In particular, the printer prints lists by recursive descent and does not attempt to detect circularities.

The *#,* reader macro, specifying *load-time* evaluation is not functionally distinct, in this release, from the *#.* reader macro. That is, *#,* currently evaluates the form that it precedes at *read time.*

4 Miscellaneous features

The Tektronix 4406 does not maintain time with respect to Greenwich Mean Time (GMT), whereas COMMON LISP's time functions require that the time be reported with respect to GMT. The following variables may be set when Tek COMMON LISP is built or they may be set in one's *.clinit.cl* file so that the COMMON LISP time functions will work correctly.

time-zone *[Variable]*

■ This variable is set to the number of hours west of GMT (ignoring Daylight-Savings Time). The values for time zones in the continental United States are shown below.

Time zone	**time-zone**
Eastern	*5*
Central	*6*
Mountain	*7*
Pacific	*8*

local-seconds-west-of-gmt *[Variable]*

■ This variable should be set to (* *time-zone* 3600) since there are 3600 seconds in one hour.

daylight-savings-time-observed-p *[Variable]*

■ This variable should be *t* if Daylight-Savings Time is observed during the summer months.