LISP LORE: A GUIDE TO PROGRAMMING THE LISP MACHINE

Hank Bromley

LISP LORE: A GUIDE TO PROGRAMMING THE LISP MACHINE

AT&T

LISP LORE: A GUIDE TO PROGRAMMING THE LISP MACHINE

by

Hank Bromley
AT&T Bell Laboratories

KLUWER ACADEMIC PUBLISHERS
Boston/Dordrecht/Lancaster

Distributors for North America:
Kluwer Academic Publishers
101 Philip Drive
Assinippi Park
Norwell, Massachusetts 02061, USA

Distributors for the UK and Ireland:
Kluwer Academic Publishers
MTP Press Limited
Falcon House, Queen Square
Lancaster LA1 1RN, UNITED KINGDOM

Distributors for all other countries:
Kluwer Academic Publishers Group
Distribution Centre
Post Office Box 322
3300 AH Dordrecht, THE NETHERLANDS

Library of Congress Cataloging-in-Publication Data

Bromley, Hank.
  Lisp lore.

  Includes index.
  1. LISP (Computer program language) I. Title.
QA76.73.L23B75 1986   005.13'3   86-7377
ISBN 0-89838-220-3

TABLE OF CONTENTS

LIST OF FIGURES

PREFACE

This book had its genesis in the following piece of computer mail:

> From allegra!joan-b Tue Dec 18 09:15:54 1984
> To: sola!hjb
> Subject: lispm
>
> Hank, I've been talking with Mark Plotnik and Bill Gale about asking you to conduct a basic course on using the lisp machine. Mark, for instance, would really like to cover basics like the flavor system, etc., so he could start doing his own programming without a lot of trial and error, and Bill and I would be interested in this, too. I'm quite sure that Mark Jones, Bruce, Eric and Van would also be really interested. Would you like to do it? Bill has let me know that if you'd care to set something up, he's free to meet with us anytime this week or next (although I'll only be here on Wed. next week) so we can come up with a plan. What do you think? Joan.

(All the people and computers mentioned above work at AT&T Bell Laboratories, in Murray Hill, New Jersey.) I agreed, with some trepidation, to try teaching such a course. It wasn't clear how I was going to explain the lisp machine environment to a few dozen beginners when at the time I felt I was scarcely able to keep *myself* afloat. Particularly since many of the "beginners" had PhD's in computer science and a decade or two of programming experience. But the need was apparent, and it sounded like fun to try, so we had a few planning sessions and began class the next month.

From early January through late March we met once a week, about a dozen times in all, generally choosing the topic for each session at the conclusion of the previous one. I spent the last few days before each meeting throwing together lecture notes and a problem set (typically finishing shortly *after* the announced class time). By the end of the course, the students had attained varying levels of expertise. In all likelihood, the person who learned the most was the instructor; nothing provides motivation to figure something out like having committed oneself to talking about it.

After it was over, another co-worker saw the sizable pile of handouts I had generated and proposed that it would make a good book. He offered to contact a publisher he had recently dealt with. I was at first skeptical that the informal notes I had hurriedly concocted would interest a reputable academic publisher, but after taking another look at the materials that had sprouted, and discussing the matter, we agreed that quite a few people would find them valuable. I've spent the last few months filling out and cleaning up the pile, and Presto, change-o. My "set of handouts" is "a book."

There are a number of people who have, in one way or another, consciously or otherwise, helped create this book. Ken Church was instrumental in arranging my first experience using the lisp machine, and later was responsible for bringing me to Bell Labs. He also taught a course here, before I came, which laid some of the groundwork for my own course. Eva Ejerhed, in a rare act of faith, hired me to work on a lisp machine thousands of miles from the nearest expert assistance, without my having ever touched one. Joan Bachenko and Bill Gale first suggested I teach a course at the Labs. Many of my colleagues who served as experimental subjects by participating in one of the three trials of the course provided useful comments on the class handouts; among those whose contributions I particularly recall are Mark Liberman, Jeff Gelbard and Doug Stumberger. Ted Kowalski first broached the idea of making a book from the handouts, and also — with Sharon Murrel — supplied lots of assistance with the use of their *Monk* text formatting system. Wayne Wolf suggested improvements to my coverage of managing multiple processes. Jon Balgley, of Symbolics, Inc.,* wrote a helpful review of one

* Symbolics, Symbolics 3600, Symbolics 3640, Symbolics 3670, and Document Examiner are trademarks of Symbolics, Inc.
Zetalisp® is a registered trademark of Symbolics, Inc.

version of the manuscript. Valerie Barr introduced herself to the lisp machine by actually working through an entire draft, making a great many valuable observations along the way. Mitch Marcus and Osamu Fujimura, my supervision at the Labs, were most understanding about the amount of time I put into this project. Carl Harris was an obliging and patient Publisher. Finally, Symbolics, Inc. graciously allowed me to quote extensively from their copyrighted materials, and Sheryl Avruch of Symbolics made possible the distribution of a tape to accompany this book.

I would like to hear about any problems readers have while working their way through the text. Please don't hesitate to mail me any of your comments or suggestions.

Hank Bromley
December, 1985

computer mail:	US mail:
hjb@mit-mc (arpa)	
	AT&T Bell Laboratories, room 2D-410
alice	600 Mountain Avenue
research } !sola!hjb (uucp)	Murray Hill, NJ 07974
allegra	

LISP LORE: A GUIDE TO PROGRAMMING THE LISP MACHINE

INTRODUCTION

The full 11-volume set of documentation that comes with a Symbolics lisp machine is understandably intimidating to the novice. "Where do I start?" is an oft-heard question, and one without a good answer. The eleven volumes provide an excellent reference medium, but are largely lacking in tutorial material suitable for a beginner. This book is intended to fill that gap. No claim is made for completeness of coverage — the eleven volumes fulfill that need. My goal is rather to present a readily grasped introduction to several representative areas of interest, including enough information to show how easy it is to build useful programs on the lisp machine. At the end of this course, the student should have a clear enough picture of what facilities exist on the machine to make effective use of the complete documentation, instead of being overwhelmed by it.

The desire to cover a broad range of topics, coupled with the necessity of limiting the amount of text, caused many items to be mentioned or referred to with little or no explanation. It's always appropriate to look up in the full documentation anything that's confusing. The manuals are perfectly adequate reference materials, as long as you know what you want to look up. The point in *this* text is rarely to explain what some specific function does in isolation — that's what the manuals are good for. The focus here is on how to integrate the isolated pieces into real applications, how to find your way around the landscape, how to *use* the multitudinous features described in such splendid detail in the 11 volumes. The manuals provide a wonderfully thorough, but static, view of what's in the lisp machine environment; I've tried to provide a dynamic view of what that environment looks like in action, or rather in interaction with a human.

The book assumes some background in lisp; the reader is expected to have experience with some dialect of the language. If you lack such experience, you may want to do a bit of preparatory study.* This course concentrates on those aspects of lisp machine lisp ("Zetalisp") which are not found in most dialects, and on the unique overall programming environment offered by the lisp machine. No experience with the lisp machine itself is assumed.

Finding an ideal order of presentation for the various topics would be difficult. Many topics are interdependent, such that knowing either would help in figuring out the other. Presenting them simultaneously would only confuse matters, so I've had to settle on one particular linear sequence of topics. It may seem natural to some readers and bizarre to others. I've tried to identify places where it might be helpful to look ahead at sections further on in the text, but I'm sure I haven't found them all, so don't hesitate to engage in a little creative re-ordering if you feel the urge. One chapter whose position is problematic is that on flavors. Conceptually, it is probably more difficult than both of the two subsequent chapters (*More on Navigating the Lisp Machine* and *Flow of Control*). But I've chosen to put it first because the flavor system is extremely characteristic of lisp machine programming, making it important to discuss as soon as possible. The main barrier to mastering the lisp machine is absorbing its gestalt, much of which is implicit in the flavor system; covering that right at the beginning helps to set the tone for what follows. But if you find flavors a bit much, feel free to look through *Navigating* and maybe *Flow of Control* and come back to it.

I've adopted a rather informal tone for most of the text: people learn better if they're relaxed. Just let me caution you that "informal" doesn't mean "sloppy." There are few extra words. Lots of information is present in only one place, and apparent only if you read carefully. If you get fooled by the informality into thinking you can scan half-attentively, you'll miss things.

It must be emphasized that learning to use the lisp machine is more a matter of learning a way of thinking than of learning a set of specific programming constructs. No amount of time spent studiously poring over documentation can yield the benefits gained from sitting at a console and exploring the environment directly.

* Two widely available sources you may find well worth your time are *Lisp* (2nd edition), Winston and Horn, Addison-Wesley, 1984, and *Structure and Interpretation of Computer Programs*, Abelson and Sussman, MIT Press, 1984.

Time spent examining various parts of the system software with no particular goal in mind is anything but wasted. Once one has a feel for how things are done, an overview of how things fit together, the rest will follow easily enough. Most lisp machine wizards are self-taught; the integrated nature of the environment, and ready access to the system code, favors those who treat learning the machine as an interactive game to play.

With that in mind, a word or two of advice on the problem sets. Don't get too wrapped up in finding the "right answer." Many of the problems are, shall we say, "challenging;" they require knowledge not found in the text (and in some cases not even found in the manuals). You will need to investigate, often without knowing exactly what you're looking for. If the investigation fails to yield immediate results, I strongly recommend that rather than head straight for my solutions, you continue to investigate. Stick it out for a while, even if you don't seem to be getting much closer. You can't learn to speak a foreign language by consulting a dictionary every time you need a word you don't know — forcing yourself to improvise from what you do know is the only way. Floundering is an unpleasant but absolutely necessary part of the process, arguably the only part during which you're really learning. Similarly, you can't become a lisp wiz just by assiduously studying someone else's code. Although seeing how an experienced programmer handles a problem is certainly useful, it's no substitute for struggling through it yourself. The problem sets are largely a ruse to get you mucking around on the machine. I don't really care if you solve them, as long as you come up with some ideas and try them out with an open mind.

The examples in the text (barring typos) are known to work in Release 6.1 of the Symbolics software for the 3600 family of machines. Only the "moving icons" example requires additional support software not included in the text. That software is available on a cartridge tape, which also contains all the code for the "graph," "tree," and "moving icons" examples, as it appears here, and all problem solutions which are too long to reasonably be manually copied from the text.

To order a copy of the tape, write to the following address (you may wish to use the order form at the back of this book) and include a check for $40 made out to Symbolics, Inc. Instructions for loading the tape will accompany it.

Software Release
Symbolics, Inc.
11 Cambridge Center
Cambridge, MA 02142

Chapter 1

GETTING STARTED ON THE LISP MACHINE

1.1 The Keyboard

Note that there are many keys which don't appear on a standard keyboard. Much of what you need to know to start using a Lisp Machine boils down to knowing what the various funny keys do.

Apart from the keys for the standard printing characters (white labels on grey keys), there are two kinds of special keys. The beige keys with grey labels (shift, control, meta, super, hyper, and symbol) are all used like the shift key — you hold them down while striking some other key. These *modifier* keys may be used singly or in combination. So "control-meta-K" means type K while holding down control and meta. There is a standard set of abbreviations for the various modifier keys. They're all just what you'd expect except that the abbreviation *s* stands for *super* rather than *shift*. *Shift* is abbreviated *sh*.

The beige keys with white labels are special function keys, and are typed like standard printing characters. That is, "Select-E" means to strike Select and then strike E. And "Select c-L" means to strike Select and then hold down control and strike L.

Use the *Help* key a lot. The information it supplies depends on the context, but it usually tells you what sort of input is wanted by the program you're typing to.

You can think of the Lisp Machine as a collection of *processes*, analogous to the different users on a time-sharing system. Each process is a program written in lisp and running in a common environment which all the processes share. A process typically (but not necessarily) has a *window* for user interaction. The *Select* key is the easiest way to switch among processes. To find out what your options are, type Select-Help. The display shows you that, among other programs that may be reached in this way, you can get a lisp listener by typing Select-L, and an editor by typing Select-E. This list is by no means fixed. Users may add their own programs to the list quite easily. Here are brief descriptions of the programs that are already in the select list on a freshly booted lisp machine:

λ	Common Lisp	a (Common Lisp) lisp listener [λ = symbol-sh-L]
C	Converse	convenient way to send and receive messages from users currently logged-in on other machines (lisp or otherwise)
D	Document Examiner	a utility for finding and reading online documentation; everything in the 11-volume manual is available here
E	Editor	the powerful Zmacs editor, like Emacs plus much more
F	File System Maintenance	various display and maintenance operations on the file system of the lisp machine or of other machines
I	Inspector	structure editor for displaying and modifying lisp objects
L	Lisp	a (Zetalisp) lisp listener
M	Zmail	a comprehensive mail-reading and sending program, using many pieces of the Zmacs editor
N	Notifications	displays a list of all "notifications" (messages from running programs) you've received
P	Peek	displays the status of various aspects of the lisp machine
T	Terminal	use the lisp machine as a terminal to log in to other computers
X	Flavor Examiner	convenient way to find out about different flavors (active objects), their message-handlers, and their state variables.

The *Function* key, like Select, dispatches off the following keystroke. Function-

Help displays a list of the options. The most commonly used are Function-F ("finger"), to find out who's logged in on the various machines, Function-H ("hostat"), for a quick look at the status of all the hosts on the local Chaosnet, and Function-S to select a different window. The exact behavior of many of the Function options is controlled by an optional numeric argument; you pass the argument by pressing one of the number keys after the Function key and before the chosen letter, *e.g.*, Function-0-S.

The *Suspend* key generally causes the process you are typing to to enter a "break loop", that is, the state of its computation is suspended and a fresh read-eval-print loop is pushed on top of the current control stack. The *Resume* key will continue the interrupted computation. Suspend takes effect when it is read, not when it is typed. If the program isn't bothering to check for keyboard input, pressing Suspend will do nothing.

c-Suspend does the same thing as Suspend, but always takes effect immediately, regardless of whether the program is looking for keyboard input.

m-Suspend, when read, forces the process at which you type it into the debugger. The debugger is another story (see below), but when you're done looking around you can continue the interrupted computation with Resume.

c-m-Suspend is a combination of c-Suspend and m-Suspend. It immediately forces the current process into the debugger.

The *Abort* key is used to tell a program to stop what it's doing. The exact behavior depends on what program you're typing to. A lisp listener, for instance, will respond by throwing back to the nearest read-eval-print loop (the top level or an intervening break loop). Like Suspend, Abort only takes effect when read. If the program isn't waiting for keyboard input, you need to use *c-Abort* instead.

m-Abort, when read, throws out of all levels and restarts the top level of the process. *c-m-Abort* has this effect immediately.

The debugger prompt is a small right-pointing arrow. Once you have that, all kinds of commands are available for moving up and down the stack, getting information about the different frames on the stack, restarting execution with or without modifying arguments and variable values, etc. Try the Help key and see what you can find out. Besides all the special commands, any normal text you type will be evaluated by the lisp interpreter.

1.2 Typing to a Lisp Listener

A *lisp listener* is a window with a lisp interpreter running in it. It reads a lisp expression from the keyboard, evaluates it, prints the returned value(s), and waits for another expression. Booting a machine leaves you in a lisp listener. Whenever you're not in a lisp listener you can get to one by typing Select-L.

While waiting for input, lisp listeners usually display "`Command:`" as a prompt. The presence of this prompt indicates that the *Command Processor* (CP) is active; it provides a convenient interface to many frequently called lisp functions. (The name of a CP command won't necessarily be the same as the name of the corresponding lisp function.) CP commands don't use the same parentheses syntax as lisp expressions do. You simply type the name of the command (one or more words) followed by any arguments to the command, and finish with the Return key. But you needn't type the name of the command in its entirety — all that's required is enough to uniquely identify which command you mean. The CP command `Help` (*i.e.*, type out the letters h, e, l, p, and hit Return) lists all the defined commands. Pressing the Help key while partway through a command will display a list of only those commands which match your input thus far. Section 3.2 in volume 1 of the documentation describes all the CP commands present in the software distributed by Symbolics. You can define more of your own. One command which may be particularly valuable to new users is `Show Documentation`. You specify some topic you want looked up in the manuals and it displays a facsimile of that portion of the documentation on your screen.

You may be wondering how the command processor knows whether you intend your typein to be interpreted as a CP command or as a lisp expression. If you begin with a letter, it assumes you're starting a CP command; with a non-alphabetic initial character it tries to parse your input as a lisp expression. Since lisp expressions usually begin with a left paren, it guesses correctly most of the time. But what if you want to evaluate a lisp symbol — if the symbol's name begins with a letter, the command processor will guess wrong and look for a command with that name. The solution here is to type a comma before the symbol's name. The comma has special meaning for the command processor: it forces whatever follows to be interpreted as a lisp expression, regardless of what the initial character is.

If you'd like to know about some other features that are available whenever you're typing to a lisp listener and you don't already feel as though you've seen more than you can possibly remember, I suggest looking ahead at the section "The Input Editor and Histories" in chapter 3. It'll make life much easier as you take on the first few problem sets.

1.3 Logging In

To login, you simply use the CP command `Login` with an argument of your user-id. In my case, it looks like `Login hjb`. Alternatively, you could apply the lisp function "login" to your user-id: `(login 'hjb)`. The effect is the same, but the former requires less typing (because of the automatic command completion). The only reason I sometimes use lisp functions when there is an equivalent CP command is force of habit: the command processor is a fairly new feature, while my fingers have been typing the lisp functions for years.

It's important to keep in mind the difference between a local login to the lisp machine, and remote logins to other machines being used as file servers. Local logins are controlled by a database called the namespace. To login locally with a certain user-id requires that there be an entry in the namespace for that user-id. It does not require a password, as there is no internal security on the lisp machine.

Many things on the lisp machine can be done with no one logged in. Some operations, however, do require that someone be logged in. Modifying the namespace, for instance, is one of these operations. How, then, you may ask, do you create a namespace entry for yourself if you can't modify the namespace unless you're logged in, and you can't log in unless you're in the namespace? One option would be to log in as someone else so you can create a namespace entry for yourself, and then log in as yourself. But nothing so underhanded is really necessary. All lisp machines have a dummy user in the namespace which the system itself uses when it needs to do something which requires having someone logged in. (This situation arises most notably while the machine is being booted — no one can log in until it's finished booting, but it can't finish booting until it does a bunch of things that require someone being logged in.) The dummy user is typically named "Lisp Machine", with user-id "Lispm" or "NIL". Whatever it's called on your machine, you can always use it by typing `(si:login-to-sys-host)`. This is often a handy trick to know about. You can now edit the namespace — use the CP command `Edit Namespace Object` — and create a user object for yourself. There is introductory documentation on the namespace and the namespace editor in chapter 11 of volume 1.

Whenever you log in to a lisp machine, unless you explicitly specify otherwise, it tries to find your personal initialization file and load it into the lisp environment. This is a file containing any number of lisp forms which customize the machine for you. They will typically set some variable values and load some other files. Where the machine looks for your init file depends on what you specified for your *home host* in your namespace entry. If you specified a host running the UNIX* operating system, it will first look for a file named `lispm-init.bn` in your directory on that machine. If your home host is a lisp machine, it'll look for the newest version of a file named `lispm-init.bin`.

The issue of remote logins arises whenever you try to do something from the lisp machine on another computer across a network, like read or write a file. If the remote host is a lisp machine, it won't ask for a password, and your local machine can take care of establishing the connection with no intervention on your part. But if the remote host is the sort that believes in security, say a UNIX system, it'll stop your local machine from doing anything until you provide an appropriate login id and password. Your local machine will pass the request right along to you. But it's essentially a matter between you and the remote host — the local machine doesn't care what username you use on the remote machine, or whether it's one that exists in the namespace. The local machine is just a messenger in this case. It will, however, try to be helpful. If you specify in your namespace entry what usernames you want to use on the various remote hosts, the local machine will try those first, even if those names are arbitrary nonsense as far as the local machine can tell. And you can always override the default usernames.

And while we're on remote file systems, there's the question of whether you should keep your files on a lisp machine or some other sort of file server. It depends on what sort of set-up you have — how much disk space in what places, how many users, etc. It's often the case that you'll want to keep as few files as possible on the lisp machine disks, because the available space would be better utilized by virtual memory and saved lisp worlds. Files can be kept on any machine you have an ethernet connection to, with little loss of efficiency, so large file systems are effectively dead weight on a lisp machine. If at all practical it makes more sense to convert every available megabyte to virtual memory or room for saved worlds, which have to be on the local disk.

1.4 The FEP

Having looked at the disk label brings up the Front-End Processor, or *FEP*. The Fep is a 68000-based computer which handles starting up the main processor of the lisp machine, and may also become active if the lisp machine enters an illegal state, whether because of a hardware malfunction or system-software bug. You can tell if your lisp machine is in the Fep if there's a prompt that looks like this: "`Fep>`", and the clock at the lower left of the screen has stopped. For now, there are just three Fep commands you should know. *Continue*, which may be shortened to

* UNIX is a trademark of AT&T Bell Laboratories.

"con", tells the Fep to try having the lisp machine resume exactly where it left off. If you were thrown into the Fep because of some serious system error, this is not likely to work — you will probably be thrown right back into the Fep. But not always. *Start*, which may be shortened to "st", does a *warm boot*. It tries to re-start all of the machine's active processes while preserving the state of the lisp environment (*i.e.*, function and variable bindings). This is a something of a kludge.* It can put things into an inconsistent state, and is something of a last resort, but it is sometimes the only way to get a wedged machine going again, short of wiping the environment clean, and losing whatever work was in progress. That is the effect of the Fep command *Boot*, which may be shortened to "b". Boot does a *cold boot*. It clears the machine's memory, reloads the microcode, restores one of the saved worlds into the virtual memory, and does a start. This is how you get a fresh machine.

There are times when you may want to get into the Fep so you can do a warm or cold boot. Perhaps your machine has been used by someone else who has significantly changed the lisp environment in unfamiliar ways, or who has used up nearly all your virtual memory. Then you will likely want to do a cold boot. Or perhaps, as often happens to me, you were playing with some critical part of the system code (after all, it's written in lisp and is completely accessible), did something unwise, and now your machine is wedged, responding neither to keyboard input nor mouse clicks. Then you may wish to resort to a warm boot, and salvage what you can. So to get into the Fep, the preferred method is to use the CP command `Halt Machine` [or evaluate `(si:halt)`]. That'll do it. But if your machine isn't responding to the keyboard, typing a command isn't an option. Then you'll have to use hyper-control-Function. Yes, if you hold down hyper and control and type Function, your lisp machine will enter the Fep under any conditions other than hardware failure. This is not the preferred method because the lisp processor will be rather rudely interrupted and may leave things in an inconsistent state. But if all else fails, it is the appropriate action.

---

* **KLUGE, KLUDGE** (*klooj*) *noun*.

1. A Rube Goldberg device in hardware or software.
2. A clever programming trick intended to solve a particularly nasty case in an efficient, if not clear, manner. Often used to repair BUGS. Often verges on being a CROCK.
3. Something that works for the wrong reason.
4. *verb*. To insert a kluge into a program. "I've kluged this routine to get around that weird bug, but there's probably a better way." Also "kluge up."
5. A feature that is implemented in a RUDE manner.

(*The Hacker's Dictionary*, Guy L. Steele, Jr., *et al*, Harper & Row, Publishers, New York, 1983.)

1.5 Random Leftovers: the mouse, the monitor, the editor

A few observations on the mouse. The functions associated with clicking the mouse buttons are completely context-dependent. It's up to the window the mouse is over when you click. It is generally the case, though, that the current binding of the buttons will be documented in the reverse video line near the bottom of the screen. And it is also generally the case that clicking once on the left button will select the window the mouse is pointing to, and clicking twice on the right button will get you to the *system menu*. The system menu offers many useful operations, such as creating windows, moving and reshaping existing windows, selecting some of the programs which are accessible via the Select key, and some which are not, etc. Play with it. It's a very good habit to keep an eye on the mouse documentation line.

There is also a lot of other useful information available at the bottom of the screen. From left to right, we have the date and time; the user-id of the currently logged in user, if any; the current *package* (the set of all symbols is partitioned into packages, to minimize name conflicts — a cold-booted machine starts out in the "user" package, which is where you'll probably do most of your work at the beginning); the state of the current process ("`Tyi`" means awaiting keyboard input); and all the way on the right, the names of any files that are open for reading or writing, or a notice of what services have been invoked locally by some other machine. And underneath the line of text you can sometimes see a few thin horizontal lines. These are the *run bars*. The one immediately under the process state goes on when some process is actively running. The one a bit to the left of that, midway between the process state and the current package, indicates that you are paging, waiting for something to be brought in from disk. The other two run bars, which appear under the current package, you will see less often. They are related to garbage collection.

One last bit of information on the monitor. We have some lisp machines which are 3600 models, and some which are newer 3670s. The two models are quite close in most respects. One way in which they differ is how the brightness of the display is controlled. The 3600 monitors have a knob on the bottom side of the console cabinet, in the front right corner. To adjust the brightness of a 3670, hold down the Local key and press "B" for brighter or "D" for dimmer. (3640s are like 3670s in this respect.)

The Zmacs editor. The built-in editor commands are multitudinous, and the total number of available commands is continually growing because it's fairly easy and very tempting to add new ones. The Appendix lists the most basic commands, but by far the best way to find out what's around is to get used to using the online documentation. Some aspects of the lisp machine can be mastered by reading the manuals, but the editor is not one of them. Type *Help* to an editor window, and type *Help* again. Start exploring. The most commonly helpful of the help options are A (Apropos), C (Command), and D (Describe). To get started, use *Help-C* on c-X c-F and on c-X c-S. You should also try *Help-A* on "compile." And one of the best sources of information on the lisp machine is the m-. command (meta-period). It prompts for the name of something, then finds the file where whatever you typed is defined. The something is often a function, but it can also be many other kinds of lisp objects: a global variable, a flavor, a resource... Two other very useful features of the editor that you might not run into right away are these: if you type *Suspend*, you get a lisp listener which starts at the top of the screen and grows as you need it. This funny window is called the typeout-window. *Resume* returns to the editor. And m-X Dired, which is also invoked by c-X D, is a utility for editing directories. Call it on some directory and type *Help*. (Keep in mind that if it's a lisp machine directory, there's no security to keep you from deleting absolutely anything.)

1.6 Problem Set #1

Questions

This "problem set" is really just a sample programming session, to familiarize you with basic operations on the lisp machine.

1. Create a namespace entry for yourself.
2. Log in.
3. Switch to the editor and create an empty buffer for a file in your home directory named "fact.lisp" (if your hɔme directory is on a lisp machine) or "fact.l" (if your home directory is on a UNIX machine with a 14 character limit on file names).
4. Enter the text for a function named "fact" which returns the factorial of its argument.
5. Compile the function from the editor with c-sh-C, and test it from the typeout window.
6. When you're satisfied with the function's performance, save the buffer and compile the resulting file.
7. Cold boot the machine.
8. Log in, and note that the function "fact" is now undefined.
9. Load the compiled version of your file.
10. Run your function successfully.

Solutions

1. `(si:login-to-sys-host)`, then `Edit Namespace Object`. Click on "create," click on "user," enter your chosen login-id, fill in the required fields (marked with *), if you wish fill in the optional fields, click on "save," click on "quit."
2. `Login yourid`
3. Select-E, c-X c-F `fact.lisp` (or `fact.l`)
4.
```
(defun fact (n)
  (if (zerop n)
      1
      (* n (fact (1- n)))))
```
5. Type c-sh-C while anywhere inside the text of the function to compile it. Then hit the Suspend key to get to the typeout window, and evaluate `(fact 5)`. The Resume key returns to the editor window.
6. c-X c-S, Meta-X Compile File (Actually, if you skip the c-X c-S, Compile File will ask if you want the buffer saved first.)
7. Suspend or Select-L, then `Halt Machine`. Type `b` (then Return) to the Fep prompt.
8. `Login yourid`, then `(fact 5)`
9. `(load "host:>dir>subdir>fact")` for a lisp machine, `(load "host://dir//subdir//fact")` for a UNIX host.
10. `(fact 5)`

Chapter 2

WHAT'S A FLAVOR?

(For a more detailed presentation of this material, see Part X, Flavors, in volume 2 of the Symbolics documentation. I have skipped many features of flavors which you may find useful, and which are fully described there.)

The flavor system is the lisp machine's mechanism for defining and creating active objects, that is, objects which can receive messages and act on on them. A *flavor* is a class of active objects. One such object is called an *instance* of that flavor.

There are two primary characteristics of a flavor: the set of messages an instance of that flavor can receive, and the set of state variables an instance of that flavor has. The state variables are called *instance variables*. Every object of a given flavor has the same set of instance variables, but the values of those instance variables are likely to vary from object to object. And for each message a flavor can receive, it has a corresponding function to invoke. The function which gets called to handle a particular message is called the flavor's *method* for that message. That method is shared by all instances of the flavor.

So, for instance, the window you see on a freshly booted machine is an instance of the flavor **tv:lisp-listener**. Like any instance of lisp-listener, it can handle 286 different messages (as of the current software). One of the messages it handles is `:expose`. Its expose method is a function which makes the lisp-listener visible on your screen, if it is not already. All lisp-listeners have the same expose method. One of the instance variables of flavor lisp-listener is `exposed-p`. All lisp-listeners have an exposed-p instance variable. If a given lisp-listener happens to be exposed, perhaps because you just sent it the :expose message, the value of its exposed-p instance variable will be `t`. Otherwise it will be `nil`.

2.1 Basic Usage

Flavors are defined with the **defflavor** special form. Here is a simple definition of a flavor named "ship," which might be used in a program for a space wars game.

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

It states that all instances of flavor ship will have five instance variables, as listed. (The empty list following the instance variables is related to a feature we'll consider in the section "Mixing Flavors.") Here are two methods for the ship flavor, to handle the messages `:speed` and `:direction`.

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

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

A **defmethod** looks very much like a **defun**. It has a function-spec, an argument list, and a body. The body will be executed in an environment in which the names of ship's instance variables will refer to the instance variables of the specific ship object which received the message.

We might also wish to have methods which allow one to examine the values of ship's instance variables. Like:

```
(defmethod (ship :x-position) ()
   x-position)
```

Writing one of these methods for every instance variable would be tedious. Fortunately there is an option to defflavor that automatically generates such methods for all the instance variables. There is also an option which causes defflavor to automatically generate methods for setting the values of all the instance variables. Their definitions are as though one had done:

```
(defmethod (ship :set-x-position) (new-position)
   (setq x-position new-position))
```

To get both of these options, our updated call to defflavor would look like:

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

To make an instance of flavor ship, we use the **make-instance** function:*

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

This will return an object whose printed representation looks like `#<SHIP 25564553>`. (The funny number will be the virtual memory address, in octal, of the instance.)

To send a message to an instance, you use the **send** function. (The effect of send is in fact identical to that of **funcall**, but when funcalling an instance send is preferred for reasons of clarity.) We can now do things like:

```
(send my-ship :set-x-velocity 1000)
(send my-ship :set-y-velocity 500)
(send my-ship :speed)        →        1118.0339
```

In addition to the instance variables, another very important variable is bound during the execution of a method. The value of the variable **self** will be the instance object itself. It's often used to send the object another message:

```
(defmethod (ship :check-speed) ()
```

* in the special case where the object is a window, you should instead use **tv:make-window**, which will perform some necessary bookkeeping operations in addition to calling **make-instance** for you.

```
  (when (> (send self :speed) 3.0e8)
    (ferror "travel at rates greater than the ~
             speed of light is not permitted")))
```

2.2 Initial Values for Instance Variables

Instances of our ship flavor start out with all their instance variables unbound. Sending a newly constructed ship the `:x-velocity` message, for instance, would result in an unbound-variable error. But there are two ways to arrange for initial values to be assigned to an instance when it is made. If you have used the `:initable-instance-variables` option to defflavor, then you may specify the initial values in the call to make-instance. So with this defflavor:

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

you could use this call to make-instance:

```
(make-instance 'ship :x-position 30 :y-position -150
                     :mass 10)
```

The instance variables mentioned in the call will have the specified initial values. Instance variables not mentioned will be unbound, as before. Now suppose you want all instances to have certain initial values for certain instance variables. Perhaps you want the x-velocity and y-velocity of all new ships to be 0. You could specify so in every call to make-instance. But there is an easier way. You can specify in the defflavor what initial value you wish the instance variables to have. Here's our next version of the defflavor for ship:

```
     (defflavor ship (x-position
                      y-position
                      (x-velocity 0)
                      (y-velocity 0)
                      mass)
                 ()
       :gettable-instance-variables
```

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

Now all ships will start out with x- and y- velocities of 0 — unless you specify otherwise in the make-instance. An initial value specified in make-instance will override any default initial values given in the defflavor.

Here is a slightly more complex example, taken from p. 427 of the flavor documentation:

```
(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)

(setq another-ship
      (make-instance 'ship :x-position 3.4))
```

The values of the new ship's instance variables will be 3.4 for x-position (the make-instance specification overrides the default of 0.0), 0.0 for y-position (the default), 2.0 for x-velocity and 3.0 for y-velocity (the values of the two global variables), and mass will be unbound.

It's useful to know that **describe**, a function which tries to print helpful information about its argument, no matter what it is, lists the values of all the instance variables when applied to an instance:

```
     (describe another-ship)   would print

     #<SHIP 40113625>, 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:                       unbound
```

2.3 Mixing Flavors

The real power of the flavor system lies in its facility for producing new flavors by combining existing ones. Suppose we wished to define an "asteroid" object. It would need much of the same functionality as the ship flavor. In fact, all of ship's instance variables and methods would be appropriate. But we do want to have two distinct kinds of object, because we might wish to add more functionality to ship and asteroid which would not be shared. Ship, for instance, could use an instance variable for its engine power, or we might want to give each ship a name. And for each asteroid we might want to characterize its composition (perhaps part of the game requires replenishing resources by mining asteroids).

One way to handle the situation would be to duplicate the lisp code for the common functionality in both flavors. Such duplication would clearly be wasteful, and the program would become far more difficult to maintain — any modifications would have to be repeated in both places. A better approach would be to isolate the common functionality and make it a flavor in itself. We can call it "moving-object." Now the ship and asteroid flavors can be built on moving-object. We just need to specify the added functionality each has beyond that provided by moving-object. The defflavor for moving-object can be exactly like our existing defflavor for ship. The new ship defflavor will have moving-object specified in its list of component flavors, which up until now has been an empty list.

```
(defflavor ship (engine-power name)
                (moving-object)
  :gettable-instance-variables
  :initable-instance-variables)
```

And asteroid:

```
(defflavor asteroid (percent-iron)
                    (moving-object)
  :gettable-instance-variables
  :initable-instance-variables)
```

Ship and asteroid both inherit all of moving-object's instance variables (including their default values) and all of its methods. They are each specializations of the abstract type moving-object. And the specialization could continue. We could define a "ship-with-passengers" flavor, built on ship, with an added instance variable `passengers`, and added methods for `:add-passenger` and `:remove-passenger`.

A flavor is not limited to having only one component flavor — it may have any number. So the set of components for a given flavor is actually a tree, consisting of all the flavor's direct components, and all of their direct components, and so on. *Figure 1* shows the tree for the flavor **tv:lisp-listener**. (All are in the `tv` package, unless otherwise noted.) As you can see, it can get quite elaborate. tv:lisp-listener has 26 component flavors in all. Of the 287 handlers I mentioned earlier that tv:lisp-listener has, only one of them is locally defined in tv:lisp-listener. The rest are all inherited, about 150 from **tv:sheet** alone, and another 50 or so from **tv:stream-mixin**.

2.4 Combined Methods

Saying simply that a flavor inherits all the methods of its components sweeps an important issue under the rug. What happens if more than one of its components define methods for the same message? Which gets used? It depends on the ordering of the component flavors. As it says on p. 432 of the documentation, "The tree of flavors is turned into an ordered list by performing a top-down, depth-first walk of the tree, including nonterminal nodes before the subtrees they head, and eliminating duplicates." So the for tv:lisp-listener, the ordered list starts with: lisp-listener, listener-mixin, listener-mixin-internal, process-mixin, window...

If more than one component flavor defines a method for a given message, with the kind of methods we have seen so far, the one which appears first on the list is taken as the combined flavor's method for that message. In particular, this means that any methods (again, of the type we have seen so far) defined locally in the new flavor will supersede all methods (for the same message) defined in any of the component flavors, since the new flavor is first on the ordered list. For example, the flavors tv:lisp-listener and tv:essential-window both define methods for the message `:lisp-listener-p`. The one in tv:essential-window always returns `nil`. The one in tv:lisp-listener always returns `t`. So a window without tv:lisp-listener mixed in will answer the `:lisp-listener-p` message with `nil`. But a lisp-listener will answer `t`, because its local method overrides tv:essential-window's.

> [There is an exception to the top-down depth-first rule. If you're already confused, skip these two paragraphs for now and come back later. If not, it's time to learn about the `:included-flavors` option to defflavor. Suppose you're defining a flavor

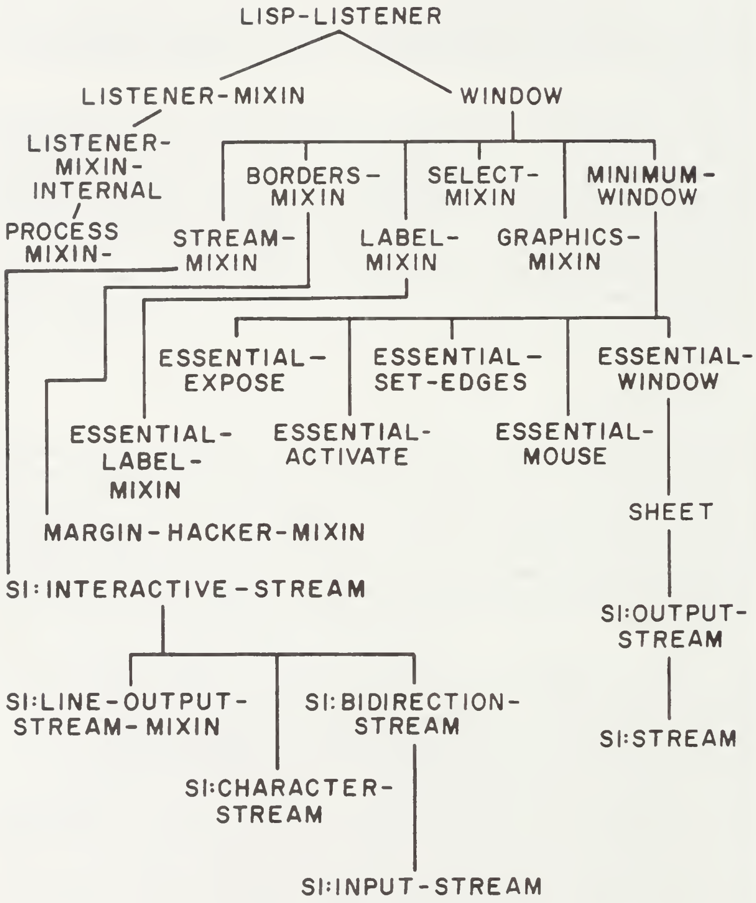


Figure 1. Flavor tree for lisp-listener

(call it my-flavor) which is intended to be used as a component for other flavors (call one of them user-flavor). And suppose that for the methods defined in my-flavor to work properly, it's necessary that some other flavor (call it needed-flavor) also be mixed into the user-flavor. And suppose further that needed-flavor is a rather basic flavor, and so should appear towards the end of user-flavor's ordered list. You could guarantee that needed-flavor always be present whenever my-flavor is present by making it a component of my-flavor. But then it would appear just behind my-flavor in user-flavor's ordered list of components. (Possibly closer to the front if some other component flavor uses it, but certainly no further back.) And some of needed-flavor's methods might override methods of other flavors which were really intended to override needed-flavor's methods, expecting to find needed-flavor near the end of the list. The solution here is the `:included-flavors` option. My-flavor lists needed-flavor as an included flavor. The effect is that when user-flavor uses my-flavor, needed-flavor will appear in the ordered list immediately after my-flavor *only* if no other flavors in user-flavor have needed-flavor as a component. If some other flavor does — and the expected case is that somebody near the end of the list will — then needed-flavor will appear there, as though my-flavor never mentioned it.

An example: look back at the lisp-listener tree, and note the position of process-mixin. It will be the fourth flavor in lisp-listener's ordered list. The defflavor for process-mixin specifies that essential-window is an "included-flavor," because for process-mixin to work properly, any flavor which uses it must also use essential-window. But if process-mixin specified essential-window as a normal component, essential-window would be brought all the way from its current position near the end of the list to fifth position, just behind process-mixin. Worse yet, sheet would be pulled from last to sixth, because it is a component (normal) of essential-window. But many of sheet's methods are supposed to be overridden by the other flavors, *e.g.*, select-mixin. If sheet were pulled in front of select-mixin, the lisp-listener would never see many of the select-mixin methods, and it wouldn't behave properly at all.]

As I've hinted, there are more kinds of methods than we have so far seen. All our methods have been what are called "primary" methods, and by default, when there is more than one primary method for the same message in the ordered list of component flavors, the one which appears first overrides all others. But sometimes you don't want to completely override the inherited primary method; sometimes you would like to specify something to be done in addition to the action of the inherited method rather than instead of. Then you would define a `:before` or an `:after` method, often called *before* and *after daemons*.

Here's how it works. Suppose I did the following defmethod:

```
(defmethod (asteroid :after :speed) ()
   (do-something-or-other))
```

Asteroid already has a primary `:speed` method, inherited from moving-object. Once this new `:after :speed` method is defined, asteroid will have a "combined method" for `:speed`, consisting of a call to the moving-object primary method followed by a call to the asteroid `:after` method. Any number of flavors in the ordered list of components may provide daemons. They will all be included in the resulting combined method. To quote from the documentation again, "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 will run before any of those in foo, and bar's after-daemons will run after any of those in foo." The primary method which appears first in the list will be called after all the before daemons (even if some of the before daemons appear later in the list than the primary method) and before all the after daemons.

The value returned by a combined method is exactly the value returned by the primary method — before and after daemons are executed only for side effect, *i.e.*, their return values are ignored. It is allowable to have before and after daemons for a message which has no primary method; in such a case the combined method will return `nil`.

Before and after daemons provide a lot of flexibility (perhaps more than you'd like to have just yet), but sometimes not enough. Fairly frequently a situation demands altering the context in which a primary method runs. A typical case would be binding a special variable to some value around the execution of the primary method, or putting the primary method into an **unwind-protect** or inside a **catch**.* Or deciding in some cases to skip the primary method altogether. Before and after daemons are unable to do any of these. The kind of method which can is called a *whopper*.

Whoppers are best explained by example. Here are three which handle the cases I

* If **unwind-protect** or **catch** are unfamiliar, you might want to look them up in volume 2.

just listed. To understand them you'll need to know that **continue-whopper** is a system-provided function which calls the regular (non-whopper) methods for this message.

```
(defwhopper (some-flavor :some-message) (arg1 arg2)
  (let ((some-special-variable temporary-value))
    (continue-whopper arg1 arg2)))

(defwhopper (some-flavor :some-message) (arg1 arg2)
  (unwind-protect
    (progn (setup)
           (continue-whopper arg1 arg2))
    (cleanup)))

(defwhopper (some-flavor :some-message) (arg1 arg2)
  (unless (some-special-test arg1)
    (continue-whopper arg1 arg2)))
```

Unlike before and after daemons, whoppers have control over the value returned by the combined method. They most commonly just pass up the value returned by continue-whopper (which will be whatever the primary method returns, as before), but they needn't. I could, for instance, do this:

```
(defwhopper (doubling-mixin :calculate) (arg1 arg2)
  (* 2 (continue-whopper arg1 arg2)))
```

And since continue-whopper is just a function like any other, there's no reason you couldn't do something like this:

```
(defwhopper (doubling-mixin-of-another-sort :some-message)
            (arg1 arg2)
  (continue-whopper arg1 arg2)
  (continue-whopper arg1 arg2))

(defwhopper (yet-another-doubling-mixin :some-message)
            (arg1 arg2)
  (continue-whopper arg1 (continue-whopper arg1 arg2)))
```

One point about ordering needs to be clarified. A whopper surrounds not just the primary method, but all the before and after daemons, too. So suppose flavor "out" is built on top of "in," and both out and in have a whopper, a before daemon, an after daemon, and a primary method for message `:mumble`. Out's combined method for `:mumble` would look like *Figure 2.*

```
     .
     .
     .             .
     .             .
     .             .       out-before
     .             .
     .             .       in-before
     .             .
out-whopper   in-whopper   out-primary
     .             .
     .             .       in-after
     .             .
     .             .       out-after
     .             .
     .             .
     .
     .
```

Figure 2. Structure of combined method

There is another kind of construct called a wrapper. This was the predecessor of the whopper, but now that the whopper exists, which is easier to use, there is seldom any need to use wrappers.

2.5 Other Ways of Combining Methods

All that I said in the previous section applied only to the default style of combining methods, called the `:daemon` type. There are actually about a dozen different types of method combination. Moreover, users can define additional types (with some effort). Before despairing, though, note that at least 90% of the time the `:daemon` type of method combination is used. All of the built-in types are discussed in chapter 53 of Symbolics volume 2, "Method Combination." I will describe one of them, the `:or` type, both because it is a relatively simple example of what can be done, and because I've actually seen it used.

First, in order to specify that you wish to use a type of method combination other than daemon, you use the `:method-combination` option to defflavor. Here is a defflavor pulled from the source code for the window system:

```
(DEFFLAVOR ESSENTIAL-MOUSE () ()
  (:INCLUDED-FLAVORS ESSENTIAL-WINDOW)
  (:METHOD-COMBINATION (:OR :BASE-FLAVOR-LAST :MOUSE-CLICK)))
```

essential-mouse is one of the flavors that appears in the lisp-listener tree given above. It's the mixin that enables a window to interact properly with the mouse. As you can see, this flavor has no instance variables and no component flavors. It has **essential-window** as an `:included-flavor`, though that has no bearing on the immediate issue of non-standard method combination. The `:method-combination` option is what concerns us. It says that the `:mouse-click` methods for all flavors built on this flavor should use `:or` style combination, and that the order of combination should be base flavor (essential-mouse) last.

So what's `:or` combination, and what does base-flavor-last mean? In `:or` combination, all the methods appearing in the ordered list of component flavors are collected, and each is called in turn. If a particular method returns a non-nil value, the remaining methods are skipped. Otherwise (the method returned `nil`), the next one is called. We just step through all the methods, stopping as soon as one returns a non-nil value. Base-flavor-last means that the essential-mouse method for `:mouse-click` will be the last one to be tried, *i.e.*, the methods will be collected and tried in the exact same order as their flavors appear in the ordered list of components. The opposite ordering may be specified with `:base-flavor-first`.

Suppose I define a flavor built (directly or indirectly) on essential-mouse, and give my flavor a `:mouse-click` method. Then whenever the `:mouse-click` message is sent to an instance of my flavor, my `:mouse-click` method will be called. If my method returns anything other than `nil`, no other `:mouse-click` methods will be called. If my method returns `nil`, then any `:mouse-click` methods defined by flavors which are components of my flavor will get a chance. If the only other `:mouse-click` method is essential-mouse's, or if all the others return `nil`, then the essential-mouse method for `:mouse-click` will be called.

The `:mouse-click` message is sent to the window under the mouse blinker whenever you press one of the mouse buttons. (The mouse process takes care of sending the message — you don't need to worry about it.) If you want to do something special when the buttons are pressed, you simply need to define an appropriate `:mouse-click` method. Now there are six different kinds of button presses (three different buttons, and single or double clicks on each button). Maybe you have something you want to do if there is a single click on the left button, but not if there is a double click on the middle button. One of the arguments to the method will tell which kind of button press there was, so your method should test the argument, and if it's the kind of button press you want to handle (single left), do whatever you had in mind, and return something non-nil, so that the other `:mouse-click` methods won't be called and possibly do something else with the single left click, interfering with your action. If it's some button press that you don't care about (double middle), return `nil` so that the other methods will have a chance to handle it.

Here's essential-mouse's method for `:mouse-click`, which is called if no one else handles the button press:

```
(DEFMETHOD (ESSENTIAL-MOUSE :MOUSE-CLICK) (BUTTONS X Y)
  (COND ((AND (= BUTTONS #\MOUSE-L-1)
              (NEQ SELF SELECTED-WINDOW)
              (GET-HANDLER-FOR SELF ':SELECT))
         (MOUSE-SELECT SELF)
         (SEND-IF-HANDLES SELF :FORCE-KBD-INPUT
                          `(:MOUSE-BUTTON ,BUTTONS ,SELF ,X ,Y)
                          T))
        ((OPERATION-HANDLED-P SELF :FORCE-KBD-INPUT)
         (SEND SELF :FORCE-KBD-INPUT
                    `(:MOUSE-BUTTON ,BUTTONS ,SELF ,X ,Y)
                    T))
        ((= BUTTONS #\MOUSE-R-1)
         (MOUSE-CALL-SYSTEM-MENU))
        (T
         (BEEP)))
  T)
```

You don't need to understand all the details to write your own `:mouse-click` methods. All you need to understand is the general format of testing the "buttons" argument and choosing some action accordingly.

2.6 Vanilla-flavor

si:vanilla-flavor is the generic flavor on which all other flavors are built. Even if your defflavor specifies no components, your flavor will still have vanilla-flavor mixed in because the flavor system does it automatically (unless you explicitly instruct otherwise with the `:no-vanilla-flavor` option to defflavor). But don't complain, because vanilla-flavor is very handy. It provides several extremely important methods. The `:print-self` method is called whenever an instance is to be printed. (The representation of the first ship instance we made, `#<SHIP 25564553>`, was actually printed on my monitor by ship's `:print-self` method, inherited from vanilla-flavor.) The `:describe` method is used by the **describe** function. (The example shown earlier, where the describe function listed all of a particular ship's instance variables, was printed by ship's `:describe` method, also inherited from vanilla-flavor.) The `:which-operations` method returns a list of all the messages handled by the object. The `:get-handler-for` method takes one argument, the name of a message, and returns the function object which is the instance's handler for that message.

See chapter 52 in the flavor documentation for the remaining vanilla-flavor methods.

2.7 Fun and Games

And from *The Hacker's Dictionary*, Guy L. Steele, Jr., *et al*:

FLAVOR *noun.*

1. Variety, type, kind. "Emacs commands come in two flavors: single-character and named." "These lights come in two flavors: big red ones and small green ones." See VANILLA.
2. The attribute that causes something to be FLAVORFUL. Usually used in the phrase "yields additional flavor." Example: "This feature yields additional flavor by allowing one to print text either right-side-up or upside down."

VANILLA *adjective.*

Standard, usual, of ordinary FLAVOR. "It's just a vanilla terminal; it doesn't have any interesting FEATURES." When used of food, this term very often does not mean that the food is flavored with vanilla extract! For example, "vanilla-flavored wonton soup" (or simply "vanilla wonton soup") means ordinary wonton soup, as opposed to hot-and-sour wonton soup.

This word differs from CANONICAL in that the latter means "the thing you always use (or the way you always do it) unless you have some strong reason to do otherwise," whereas "vanilla" simply means "ordinary." For example, when MIT hackers go to Colleen's Chinese Cuisine, hot-and-sour wonton soup is the canonical wonton soup to get (because that is what most of them usually order) even though it isn't the vanilla wonton soup.

2.8 Problem Set #2

Questions

Part I

1. Define a function of four arguments that draws a square. The args are the window, the x and y coords of the square's center, and the size (length of each side).
2. Define a flavor of window which handles a `:draw-square` message by calling your **draw-square** function. Then create a window of that flavor, being sure to keep a pointer to it, and verify that the `:draw-square` message works.
3. Make the size argument to the method optional, defaulting to the value of a global variable ***square-size***. Make the default size 100 pixels.
4. Arrange it so that clicking left on the mouse while over your window draws a square centered on the mouse position, using the default size.
5. A. Try these out — they temporarily bind the default size to 50 instead of 100, and then draw a square using the default size.

   ```
   (let ((*square-size* 50))
     (send w :draw-square 150 150))

   (let ((*square-size* 50))
     (process-sleep 300))          click left over my-window during the sleep
   ```

   Why doesn't the second work? Why isn't the binding of *square-size* to 50 being seen?

   B. There is special form called **let-globally** which will get around this problem. Look it up in the manual and use it with a **process-sleep** so that clicking left over the window will make a square whose size is controlled by the let-globally.
6. Making the default size an instance variable will get around the problem addressed in (5), and has several other advantages. We no longer clutter things up with an extra global variable. More important, with the default size an instance variable, it's possible for each instance of my-window to simultaneously have a different value for `default-square-size`.

   Redefine your flavor of window (and make a new instance) so the following works:

```
(let ((old-size (send w :default-square-size)))
  (send w :set-default-square-size 50)
  (process-sleep 300)
  (send w :set-default-square-size old-size))
```

7. What would happen if, during the **process-sleep** in the **let** in problem (6), I typed `c-Abort`? The reset of `default-square-size` back to its previous value would never happen. The **let**, which is intended to be without side-effect, would have permanently changed the value of the instance variable.

   Add something to the **let** in problem (6) so that it will be guaranteed to reset `default-square-size` to its previous value.

8. Without duplicating any code, define a new flavor of window, **doubling-window**, which behaves just like the window you've already defined, except that the squares it draws are twice as big as you ask for. That is, if you click left the square will be twice the size specified by the window's `default-square-size` instance variable, and if you explicitly send the `:draw-square` message the square will be twice the size specified by the third argument to the message.

Part II

Not too long ago, I had to track down some bugs in the speech editor I was working on, related to the ordering of the components of one of my flavors. I had switched the order of two components to fix one bug, and apparently introduced some new ones. At least, the program was misbehaving in a way it previously hadn't, and the only relevant change I'd made was the component reordering. It was a sort of misbehavior that would have been difficult to debug by interrupting the program while it was doing the wrong thing, examining the state of the world, and working back to see which message-handler was responsible for the new behavior. So I instead decided to find out what messages would be handled differently with the new component ordering, and work forward to see which of those changed handlers could be causing the odd behavior.

The Flavor Examiner (Select-X) has a facility for listing all the message-handlers of a given flavor, together with the name of the flavor the handler was inherited from. So all I had to do was list all the handlers for my two flavors, and compare them. If only one of the two handled a particular message, it wouldn't matter in which order the flavors were mixed in, since I would get the same handler either way. And if they both handled a message, but handled it with the same inherited method from a common component, the order of combination still wouldn't matter. Any messages, however, which were handled by both flavors but with different methods, would be likely suspects. Which handler my combined flavor had would indeed depend on the order of combination.

It would have been a simple task, except that the two flavors involved each had between 190 and 200 handlers. Both included the flavor **tv:window**, which has 194 handlers. Nearly all of the handlers I had to wade through were internal to **tv:window** (mainly from the flavors **tv:sheet** and **tv:stream-mixin**), and shared by my two flavors. There were only a handful of suspects, and once I found them it did not take too long to realize that my mistake had been in using **tv:window-pane** where I should have used **tv:pane-mixin**. The bugs went away. But it was a far more painful experience than it would have been if we had a few simple utilities for filtering the lists of handlers. This portion of the problem set asks you to write a few such facilities.

1. Write a function of two arguments, to be called on two instantiated objects of different flavors. It should return a list with four sublists: a list of handlers in flavor-1 and not in flavor-2, a list of handlers in flavor-2 and not in flavor-1, a list of handlers in both flavors, and a list of pairs of corresponding handlers, for messages handled by both flavors but with different handlers.

2. That's already enough to be useful, but often a flavor is not instantiated, and sometimes may not even be instantiable without further mixins. The function you've defined won't work on such flavors. Write a similar function which will take as arguments the *names* of two flavors.

3. The answer to (1), and depending on how you did it, quite possibly the answer to (2), are subject to a bug of sorts. They may decide under certain conditions that two handlers are different, when for all practical purposes they are not. Consider the following:*

---

* **FOO** (*foo*)

1. *interjection.* Term of disgust. For greater emphasis, one says MOBY FOO (see MOBY).

2. *noun.* The first metasyntactic variable. When you have to invent an arbitrary temporary name for something for the sake of exposition, FOO is usually used. If you need a second one, BAR or BAZ is usually used; there is a slight preference at MIT for bar and at Stanford for baz. (It was probably at Stanford that bar was corrupted to baz. Clearly, bar was the original, for the concatenation FOOBAR is widely used also, and this in turn can be traced to the obscene acronym "FUBAR" that arose in the armed forces during World War II.)

   Words such as "foo" are called "metasyntactic variables" because, just as a mathematical variable stands for some number, so "foo" always stands for the real name of the thing under discussion. A hacker avoids using "foo" as the real name of anything. Indeed, a standard convention is that any file with "foo" in its name is temporary and can be deleted on sight.

BAR

The second metasyntactic variable, after FOO. If a hacker needs to invent exactly two names

```
(defflavor foo () ())
(defflavor bar () ())

(defmethod (foo :silly-message) () nil)
(defmethod (bar :before :silly-message) () nil)

(defflavor flav-1 ()
           (foo bar))
(defflavor flav-2 ()
           (foo bar))
```

Flav-1 and flav-2 each have a combined method for the message `:silly-message`, and they use the same components for their respective combined methods. I would like these two combined methods to be considered the same, and fall into the third of our four sublists. But your functions may be putting such pairs of methods into the fourth sublist. If so, modify one or both of your answers to (1) and (2) to re-classify these methods accordingly.

4. Of all your answers to questions (1)-(3), select the one that seems to be the most useful, and merge it into the Flavor Examiner.

for things, he almost always picks the names "foo" and "bar."

(*The Hacker's Dictionary*, Guy L. Steele, Jr., *et al*)

Hints

Part I

1. Your function should either send the window the `:draw-line` message four times, or the `:draw-lines` message once. You can find out about the arguments to the `:draw-line` and `:draw-lines` messages by looking them up in the index to volume 7.

2. Your flavor of window should be built on **tv:window**. The `:draw-square` method will need to use the `self` variable.

   Using the window brings up a somewhat subtle issue. At times you'll need the lisp listener exposed so you can type commands. And at times you'll need your new window exposed so it can display the squares being drawn. You could, then, alternate between the two windows, but that becomes awkward. Far better is to position and shape the two windows such that they can both be exposed simultaneously. One procedure would be to first narrow the lisp listener down to the left half of the screen (choose *Edit Screen* from the system menu, then *Move Single*), then make your window with the following form, choosing an area with the mouse that doesn't overlap the narrowed lisp listener:

   ```
   (setq w (tv:make-window 'my-window :edges-from :mouse
                                      :expose-p t))
   ```

   Now you're ready to try drawing squares, with `(send w :draw-square ... )`.

3. The global variable should be defined with **defvar** or **defconst**, and initialized to 100 in the call to defvar/const. The redefined method should include in its arglist "`&optional (size *square-size*)`".

4. The window needs a `:mouse-click` method. Recall that `:mouse-click` methods use `:or` combination, so your method should return `t` if it handles the click, and `nil` otherwise.

5. A. Think about processes.

   B. **let-globally** looks just like a **let**.

6. The flavor will have an instance variable named `default-square-size`, and you'll need to use the `:settable-instance-variables` option. The new `:draw-square` method will have to access the instance variable.

7. Use **unwind-protect**.

8. Your new flavor should be built on the old one, and should have a whopper for `:draw-square`.

Part II

1. You can get a list of all the messages an object handles with `:which-operations`. And given a specific message, you can get the handler for it with `:get-handler-for`.
2. The Flavor Examiner finds the methods from the flavor name, without having an instantiated object. Find out how. As a simpler and nearly as useful alternative, note that the editor command M-x List Combined Methods (among others) can do the same thing as long as the flavor has been instantiated at least once, and track down what it does. (Try meta-. on com-list-combined-methods.) This won't help for uninstantiated flavors, but does remove the need to have a pointer to an actual instance in cases where the flavor has been instantiated.
3. If you're modifying your answer to (1), you're probably dealing with compiled function objects. Notice that the **describe** function, when applied to the compiled function object for a combined method, prints the necessary information under the title `extra info`, after `:fdefinition-location-hints`. Find out how.

   If you're modifying your answer to (2) you probably already have a data structure containing the necessary information, and you just need to make use of that information to re-classify the handlers.
4. The source code for the Flavor Examiner is in the file "sys: window; flavex."

Solutions

Part I

1.
```
(defun draw-square (window x y size)
  (let* ((half-size (// size 2))
         (x0 (- x half-size)) (x1 (+ x half-size))
         (y0 (- y half-size)) (y1 (+ y half-size)))
    (send window :draw-line x0 y0 x0 y1)
    (send window :draw-line x0 y1 x1 y1)
    (send window :draw-line x1 y1 x1 y0)
    (send window :draw-line x1 y0 x0 y0)))

(defun draw-square (window x y size)
  (let* ((half-size (// size 2))
         (x0 (- x half-size)) (x1 (+ x half-size))
         (y0 (- y half-size)) (y1 (+ y half-size)))
    (send window :draw-lines tv:alu-ior
          x0 y0 x0 y1 x1 y1 x1 y0 x0 y0)))
```

2.
```
(defflavor my-window ()
           (tv:window))

(defmethod (my-window :draw-square) (x y size)
  (draw-square self x y size))
```

3.
```
(defvar *square-size* 100)

(defmethod (my-window :draw-square)
           (x y &optional (size *square-size*))
  (draw-square self x y size))
```

4.
```
(defmethod (my-window :mouse-click) (buttons x y)
  (cond ((= buttons #\mouse-1-1)
         (send self :draw-square (- x tv:left-margin-size)
                                 (- y tv:top-margin-size))
         t)                    prevent other :mouse-click methods
                                 from being called (:or combination)
        (t nil)))              allow other kinds of clicks to fall through
```

5. A. It doesn't work because the `:mouse-click` method is called from the mouse process, and different processes each have their own variable binding stacks. The mouse process can see the global value of ***square-size*** (100) which was set by the defvar, but not the let binding (50) which is in the binding stack of your lisp listener. (There is

more information on processes in the next chapter.)

B.
```
(let-globally ((*square-size* 50))
  (process-sleep 300))
```

6.
```
(defflavor my-window
        ((default-square-size 100))
        (tv:window)
  :settable-instance-variables)

(defmethod (my-window :draw-square) (x y &optional size)
  ;; can't default the optional arg because environment
  ;; isn't set up yet - use "or" in body
  (draw-square self x y (or size default-square-size)))
```

7.
```
(let ((old-size (send w :default-square-size)))
  (unwind-protect
      (progn (send w :set-default-square-size 50)
             (process-sleep 300))
    (send w :set-default-square-size old-size)))
```

kwc-letf, a macro written by Ken Church, does just this sort of thing, and looks a lot nicer. (Its name was chosen to distinguish it from **letf**, a special form provided by Symbolics which has similar, but slightly different, effects.) The syntax is like that of **let**, except in place of the names of local variables to be bound, **kwc-letf** accepts any reference which can be understood by **setf**. It arranges for that reference to return the specified value if called within the **kwc-letf**, and for the old value to be restored upon exiting. In our case, it would be used like this:

```
(kwc-letf (((send w :default-square-size) 50))
  (process-sleep 300))
```

This produces identical compiled code, but makes the intent much clearer — you can see that I just want to evaluate the **process-sleep** in a context where `(send w :default-square-size)` would return 50. The call to **kwc-letf** macroexpands into:

```
(LET ((#:G0531 (SEND W :DEFAULT-SQUARE-SIZE)))
  (UNWIND-PROTECT
      (PROGN (SEND W ':SET-DEFAULT-SQUARE-SIZE 50)
             (PROCESS-SLEEP 300))
    (PROGN (SEND W ':SET-DEFAULT-SQUARE-SIZE #:G0531))))
```

8.
```
(defflavor doubling-window ()
           (my-window))

(defwhopper (doubling-window :draw-square) (x y &optional size)
  (continue-whopper x y (if size (* 2 size)
                              (* 2 default-square-size))))
```

An alternative whopper would be:

```
(defwhopper (doubling-window :draw-square) (x y &optional size)
  (let ((default-square-size (* 2 default-square-size)))
    (continue-whopper x y (and size (* 2 size)))))
```

Part II

1. Here is a simple-minded implementation:

```
(defun sort-handlers (object-1 object-2)
  (loop
    for message in (union (send object-1 :which-operations)
                          (send object-2 :which-operations))
    for handler-1 = (send object-1 :get-handler-for message)
    for handler-2 = (send object-2 :get-handler-for message)
    when (not handler-2) collect handler-1 into only-1
    else when (not handler-1) collect handler-2 into only-2
    else when (eq handler-1 handler-2) collect handler-1
                                       into both-and-same
    else collect (cons handler-1 handler-2)
           into both-and-different
    finally (return (list only-1 only-2
                          both-and-same both-and-different))))
```

2. A partial answer: everything you need to know about a flavor is in the list returned by `(si:examiner-compute-magic-list (get` *flavor-name* `'si:flavor))`. The extraction of the info is done by the method `:compute-all-handlers-once` of the flavor **flavex:flavor**. As for M-x List Combined Methods, the crucial function is **zwei:find-combined-methods**.

3. To follow up the **describe** hint, this is what **describe** uses:

```
(cdr (assq :fdefinition-location-hints
           (si:cca-extra-info
             (si:compiled-function-cca function-obj))))
```

So we might replace this line in my answer for (1):

```
else when (eq handler-1 handler-2) collect handler-1
                                     into both-and-same
```

with:

```
else when (or (eq handler-1 handler-2)
              (equal (foo handler-1) (foo handler-2)))
       collect handler-1 into both-and-same
```

where `(foo handler)` is:

```
(cdr (assq :fdefinition-location-hints
            (si:cca-extra-info
              (si:compiled-function-cca handler))))
```

And if you used my partial answer for (2), the list returned by **si:examiner-compute-magic-list** still contains everything you need to know, although it's possible your answer to (2) threw away some of the information needed for (3).

4. At the moment, this is an open problem — I don't know exactly what's required to make such an addition. But it sure would be nice to have...

Chapter 3

MORE ON NAVIGATING THE LISP MACHINE

The last chapter discussed an aspect of programming with the lisp language, as implemented on the lisp machine. This one is about some aspects of using the lisp machine which are more or less independent of programming on it, *i.e.*, what you might call the operating system of the lisp machine.

3.1 The scheduler and processes

Switching back and forth among the different processes can be explicitly controlled by the lisp machine programmer (read the documentation on *Stack Groups*), but almost never is. A special module called the *scheduler* generally handles this responsibility. Every 1/60th second the scheduler wakes up and decides whether the current process should be allowed to continue running, and if not, which other process should get a chance.

If the current process has been running continuously for less than a second, and wishes to continue, it is allowed to. (Note that a full second is a long time for this sort of thing, compared to other timesharing arrangements.) Or if it's been running for a second but no other process wishes to run, it is still allowed to continue. But if it's been monopolizing the machine for more than a second, and one or more other processes want to run, it's forced to take a rest while the scheduler gives the others a chance. The process chosen by the scheduler is now treated as the previous current process was: it will be allowed to run until some other process(es) wish to run and the current process either volunteers to give the others a chance, or passes the one second mark.

The way a process "volunteers to give the others a chance," or, in less emotionally-laden terms, informs the scheduler that it doesn't need to run, is with the **process-wait** function. Process-wait specifies a condition the process is waiting for. When the condition becomes true, the process is ready to run. When the scheduler decides to resume the process, the call to process-wait returns and the computation continues from there. The first argument to process-wait is a string to appear in the wholine (at the bottom of the screen) while the process is waiting. The second arg is a function and any remaining args are arguments to the function. To see whether the process is ready to continue, the scheduler applies the specified function to the specified arguments. The return value of the function is what the scheduler uses for the "condition" mentioned above. This function is often called the process' *wait-function.* Here is the process-wait which is responsible for "Tyi" appearing in the wholine most of the time:

```
      (PROCESS-WAIT "Tyi" SELF ':LISTEN)
```

This call is buried somewhere in the code windows (or anything with **tv:stream-mixin**) use for reading from the keyboard. It says that the process will be ready to continue when application of `SELF` to the argument `:LISTEN` returns non-nil. Since **funcall** is equivalent to **send** when dealing with instances (see the previous chapter), this process-wait will return when `(send self :listen)` is true. The handler for :listen just checks to see if anything is in the io-buffer, so the process which calls this process-wait will forfeit its turns in the scheduler until it has something in its io-buffer.

Now a question for the bold: what happens if an error occurs in the scheduler? It is, after all, just another piece of lisp code. And even if the scheduler code itself is bug-free, all the wait-functions are called in the scheduler, and any loser* can write a buggy wait-function. It's also the case that blinking of flashing blinkers gets done from the scheduler. (There's a *clock function list* of things to be done every time the scheduler runs, and by default the only thing on the list is blinking the blinkers.) And any loser can also write a buggy :blink method for his/her blinkers — I've certainly done it. So what happens when the scheduler runs into an

* See hacker's definition at end of chapter.

error? The scheduler has no window to use. How can the debugger communicate with you?

What happens is that the scheduler enters the debugger and uses what is called the *cold-load stream*. This is a very basic stream which completely bypasses the window system. It uses the screen as it would a dumb terminal, with no regard for the previous display contents, ignoring even window boundaries. There will be no blinker (which makes typing somewhat disconcerting) and none of the input editor commands will be active, apart from the rubout key. But you will be in a legitimate debugger, from which you can attempt to set things right. So don't panic.

Our view of scheduling is now fairly complete. The current process owns the lisp machine until it either does a process-wait, or uses up its second. When either of these occurs, the scheduler calls the wait-functions of the other processes. The first process whose wait-function returns a non-nil value gets to become the current-process. If none of them do, the old current process remains the current process. And if any errors occur while in the scheduler, the debugger uses the cold-load stream.

Fine. Now it's time to complicate things again. At any given time a process is either *active* or *inactive*. Inactive processes are not even considered by the scheduler when it looks for an alternative to the current process. Their wait-functions aren't called at all until they become active. And what makes a process active or inactive? Two of the instance variables of a process are its *run-reasons* and its *arrest-reasons*. An active process is one with no arrest reasons and at least one run reason. Otherwise (at least one arrest reason or no run reasons) the process is inactive. There are messages for looking at a process' run and arrest reasons, and for adding to or deleting from them. A program might use those messages, but an interactive user is more likely to arrest or un-arrest a process in one of the following ways (all of which end up passing those same messages, but are easier to use):

1. The system menu has options for arresting or un-arresting the process in the window the mouse is over.

2. If you click on the name of a process in Peek's display of processes, you get a menu of useful things to do to that process. Two of them are arresting and un-arresting.

3. Typing Function-A arrests the process the wholine is watching. (This is usually the selected window's process. But you can change which process the wholine watches with Function-W.) Function-minus-A un-arrests it.

Another common operation to perform on a process is to *reset* it. This is very much like typing c-m-Abort to it. It flushes everything on the process' stack and restarts it. (More exactly, it reapplies the process' initial function to its initial arguments, but you needn't understand that just yet.) You can only type c-m-Abort to a process when you can select its window, which isn't always possible, but you can reset a process anytime. The options for how to reset a process are similar to those for un-/arresting one. You can send a process the :reset message; you can use the reset option in the system menu to reset the process in the window under the mouse; you can use the menu in Peek's display of processes.

Note that all these ways of resetting, except for explicitly sending the :reset message, depend on being able to use the mouse. So if the mouse process is the one which is in trouble, they won't work. The only way out is to get a handle on the mouse process and send it the :reset message. An extremely useful fact to remember is that the value of the symbol **tv:mouse-process** is always the mouse process itself (an instance of flavor si:process). So typing this will often unwedge the mouse process: `(send tv:mouse-process :reset)`.

One final note on resetting: `(send current-process :reset)` doesn't work. (It just returns nil.) The usual method for unwinding a stack doesn't work from within that stack. To reset the current process, you need to either spawn a new process for the sole purpose of resetting your process (use **process-run-function**), or use an optional argument to the reset message: `(send current-process :reset :always)` will work.

Before long you will probably have cause to create your own processes. Take a look at Part III of Volume 8 of the Symbolics documentation when the need arises.

3.2 Windows

The entire set of existing windows is organized into several trees. The root of each tree is a *screen.* (Screens are built on tv:sheet but don't have all the other mixins that make a window.) Each window has a *superior* (towards the root of the tree) and any number (possibly 0) of *inferiors* (towards the leaves). The windows option in Peek displays all the trees (subject to a restriction mentioned below).

The state of a window may be characterized in any of several ways. The window may be *selected* or *deselected*, it may be *exposed* or *deexposed*, and it may be *activated* or *deactivated*. The terminology is confusing and unfortunate. Selection, exposure, and activation are not independent factors. In fact, they are closely tied. Before a window may be selected, it must be exposed. And before it may be exposed, it must be activated. So there are four possible states for a window: *deactivated*; activated but not exposed (usually called *deexposed*); activated and exposed but not selected (usually called *exposed*); and activated, exposed, and selected (usually called *selected*).

If a window is deactivated, the system will not keep track of it. More precisely, the window will not appear in the array which is the value of **tv:previously-selected-windows**. Many parts of the system software, including Peek, use that array when they are expected to produce a list of all windows. And a deactivated window will not appear in the inferiors list of its superior. The system will not keep any pointers to such a window, so unless you have one, the window will be garbage-collectible. (Ignore this point for now if you don't understand garbage collection.)

Once activated, a window may become exposed. Being exposed means roughly that the window has somewhere for its typeout to go. Any window which is completely visible on the screen, not even partly covered by some other window, is exposed. (It is also possible for windows to be exposed without being visible. Such a window must have somewhere for its typeout to go other than your screen — that place would be a bit-array which could later be mapped onto the screen. See sections 11.4 and 11.5, *Pixels and Bit-save Arrays*, and *Screen Arrays and Exposure*, in Volume 7. For now let's just assume that only visible windows are exposed.)

If a window which is not exposed is asked to type something out (perhaps with the :tyo or :string-out messages), it won't be able to do it, since it has no place to send the typeout. How it reacts is controlled by its *deexposed-typeout-action*, which is an instance variable of tv:sheet. It may specify, for instance, that the window should try to expose itself, or that an error should be signaled. The default value of deexposed-typeout-action, `:normal`, specifies that the process doing the typeout should enter an *output hold* state. That means it will do a process-wait (remember those?) with a wholine state of "Output Hold" and a wait-function which essentially waits for the window to become exposed:

```
(PROCESS-WAIT "Output Hold"
              #'(LAMBDA (SHEET)
                  (NOT (SHEET-OUTPUT-HELD-P SHEET)))
              SELF)
```

In addition to the usual ways of exposing the window (mentioned below), when an output hold occurs there is one extra way which becomes available. That is to type Function-Escape.

Now for selection. Although any number of windows may be simultaneously exposed, as long as they can all fit on your screen without overlapping, only one window at a time may be selected. The currently selected window is always the value of the symbol **tv:selected-window**. The selected window is the one to which keyboard input is directed. It usually has a blinking rectangular cursor in it. There must, of course, be a process running in the selected window for it to do anything with the keyboard input. If the selected window has no process, typing on the keyboard has no effect, except for special keys like Function and Select.

If we imagine the four possible window states (deactivated, deexposed, exposed, selected) occupying a spectrum, the various messages for changing the state of a window may be pictured as follows:

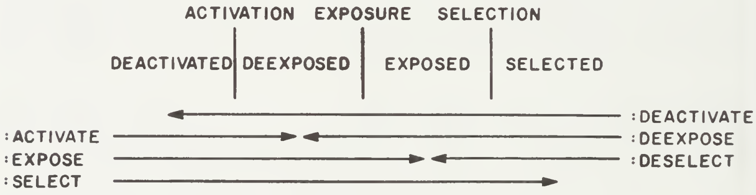


Figure 3. Transitions among window states

The meaning of the arrows is that a window sent the given message will be pushed all the way from its current state to the head of the arrow. So, for instance, if an exposed window is sent the :deactivate message, it will be both deexposed and deactivated. A selected window would be deselected, deexposed, and deactivated. The messages only push in the direction of the arrows, they don't pull. That is, if the window is already at or beyond the arrowhead nothing happens. If a selected window is sent the :activate message, there is no effect. It is not pulled back to the deexposed state.

A freshly instantiated window, as is returned by **tv:make-window**, will be deactivated, unless you specify otherwise. This is also the state of a window which has been sent the :deactivate or :kill messages. (Killing a window deactivates all of its inferiors as well as itself.)

You can always change the state of a window by sending it an appropriate message, but there are several ways to make these messages be sent without explicitly sending them yourself. The system menu has an option for killing the window under the mouse, and one for selecting a window from the list in tv:previously-selected-windows. The Edit Screen option in the system menu pops up another menu with options for killing or exposing any partially visible window, and for exposing any window in tv:previously-selected-windows. (The Edit Screen menu also has options for creating, moving, or reshaping windows.) And if you click on the name of a window while in the windows display of Peek, you get a menu with options for selecting, deselecting, exposing, deexposing, deactivating or killing the window.

There's another way to select a window which you are already familiar with: use the Select key. For the kinds of windows accessible via the Select key (Select-Help displays a list), the effect of the Select key depends on how many instances of that flavor of window exist. Let's take Select-E (for the Zmacs Editor) as an example. If there are no existing Zmacs windows, typing Select-E will create one and select it. If there is exactly one Zmacs window, Select-E will select it (unless it is already the selected window, in which case the screen will flash and it will remain the selected window). If there are more than one existing Zmacs windows, and none of them are the selected window, Select-E will select the one which had most recently been the selected window. Typing Select-E repeatedly will rotate through all the existing Zmacs windows.

Typing Select-c-E (hold down the control key while striking E) will always create and select a new Zmacs window, regardless of whether there are already some.

Windows can also be selected with the Function key. Function-S selects the previously selected window. Providing a numeric argument between the Function key and S allows rotation of the selected windows in various arcane ways. Type Function-Help and read about S for a full description.

Changing the state of a window will often cause the state of other windows to change. For instance, if I select one window, the window which had been selected necessarily becomes deselected. And if I deselect a window, some other window (the previously selected one) becomes selected. Similarly, exposing a window may partially or entirely cover some other window which had been exposed; the latter window is forced to become deexposed. And deexposing a window may uncover some other window, thereby exposing it. (A subtler point arises here. Simply sending the :deexpose message usually does not have the intended effect. Since no other windows will be covering the one which has just been deexposed, it will immediately be automatically re-exposed. It will look like nothing happened. What you probably meant to do was either expose some other window [which will automatically deexpose the first window], or send the first window the :bury message, which in addition to deexposing it, puts the window underneath all the other windows, so that the window that ends up being auto-exposed is some other one.)

The interactions among windows can become terribly convoluted. There are several kinds of locks intended to keep everything straight. If something goes wrong and an error occurs while the window system is locked, the debugger won't be able to expose a window to use. So it uses the cold-load stream, just as when an error occurs inside the scheduler.

If you've been messing with the window system in unwise ways, it's possible to get it locked up so that you can't do anything (I do it all the time). If the window which appears to be selected isn't responding to typein, and c-m-Abort doesn't help, and the mouse is dead, and you can't select some other window with the Select or Function keys, it may be that you're hung up in a locked window system. Your last resort in such a case (short of h-c-Function and a warm or cold boot) is to type Function-control-Clear Input. This clears all the locks in the window system. It's a sledgehammer, and can easily break some things, but it may revive your machine without having to boot.

One last note about windows. When a window is sent more than one screenful of typeout at a time, it may pause at the end of each screenful, type `**MORE**`, and wait for you to press any key before continuing. This behavior is called *more processing*. Whether more processing occurs (as opposed to continuous output) is controlled by Function-M and Function-c-M. Type Function-Help for details. For more processing to occur it must be turned on both globally and for the individual window.

3.3 Debugging

The *trace* feature can be invaluable in finding out why your code isn't doing what you expected. You can read all about it in chapter 4 of volume 4. The basic form works like this: you turn on tracing for a function named *foo* by evaluating `(trace foo)`. From then on, every time foo is called, a line will be printed on your screen announcing that foo has been entered and listing its arguments. And when foo finishes another line will be printed, announcing the exit from foo and listing the return value(s). You turn off tracing of foo with `(untrace foo)`. Some of the fancier features allow you to print the values of arbitrary expressions upon function entry or exit, or to make tracing conditional on some predicate, or to enter a break loop or the debugger upon entry. The syntax for these features can be difficult to remember, so I'd suggest using the trace menu to select them. (You get the trace menu by clicking on trace in the system menu or by doing Meta-X Trace in the editor.)

Trace can be used on anything you can describe with a *function spec* (see chapter 27 of volume 3). Function specs are most often symbols, but can also be something like `(:method tv:sheet :expose)`, referring to the :expose method for flavor tv:sheet.

One note of caution: you can instantaneously make your machine unusable by tracing the wrong function. If, for instance, you'd traced some function that gets called every time a character is read from the keyboard, with the feature that throws you into the debugger upon function entry, you'd have a real tough time typing "untrace." (On the other hand, the kinds of functions that can break everything when traced are not ones the new user is likely to be interested in anyway, so don't feel inhibited by this warning. Go ahead and play. You can always boot. I just thought I should mention that I've screwed myself this way more than once.)

Advising is a more general facility similar to tracing. Here you supply an arbitrary piece of code to be executed upon function entry or exit. Advice comes in three varieties: *before*, *after* and *around*. The three kinds are very much like before daemons, after daemons, and whoppers. Advising is explained in detail in chapter 4, volume 4.

The debugger itself is the main tool for discovering what went wrong. You should remember a few things about the debugger from Chapter 1: it is entered whenever an error occurs, and may be entered manually with the function `(dbg)` or by typing m-Suspend. Once in the debugger, you can move up and down the stack with c-P and c-N (for *previous* and *next*), and see the whole stack with c-B (for *backtrace*). There are generally a series of restart handlers bound to the *super* keys, and to Resume and Abort.

Now some debugger facilities that may be new. The arguments of the function in the current frame are accessible via the function **dbg:arg**. It takes one argument, specifying which of the current function's arguments you want (you may specify either by name or by number). Suppose the current function has an argument named "array," and I want to know what element #5 in the array is. I could type `(aref (dbg:arg 'array) 5)`. The best part about dbg:arg is that it can be used with **setf** to change the argument. So if the first argument to the current function were the string `"now"` and I wanted to change it to the string `"then"`, I would type `(setf (dbg:arg 0) "then")`. (Note that the numbering of arguments begins at 0, not 1.) The function **dbg:loc** is analogous to dbg:arg, but works on local variables instead of arguments.

You can also use the debugger commands c-m-A and c-m-L to get the values of arguments and local variables. Returning to the case where I wanted to see element #5 of an array which was an argument in the current frame, I could type (assuming the array were the third argument) c-2 c-m-A, which would return the array object and give me a fresh debugger prompt, then `(aref * 5)`, using `*` to access the previously returned value.

Once you've re-arranged the arguments to your liking, you may use c-m-R to reinvoke the current frame with the new arguments. Another useful command is c-R, for returning a specified value from the current frame (the value is prompted for after you type c-R).

Another handy command is c-E. It finds the definition of the function in the current frame and reads it into the editor. (Of course, it reads the whole file the function is defined in. You will be positioned in the buffer at the place where the function is defined.) Finally, c-M (for *mail*) will set you up for sending a bug report. You'll be switched to a mail window, with both a complete stack backtrace and information on what version of the system software is running on your machine inserted as the initial contents of your message. Add whatever text is necessary to describe the situation. Check the top line of the buffer to make sure the "To:" field contains the name of an appropriate recipient, and edit it if necessary. Then just hit the End key. (c-End if you have the Bell Labs/Murray Hill standard utilities — we use End for another editing function.) Off goes your mail, you get returned to the debugger, and you may continue as you please. The inclusion of the backtrace makes life much easier for your system maintainer.

3.4 Who Does What

Here are a hodge-podge of lisp functions for finding out about functions and symbols (see "Poking Around in the Lisp World," section 17.1 in volume 1, for some more): **who-calls** takes one argument, usually a symbol. It searches all defined functions and collects those which call the symbol as a function or use it as a variable or constant. This takes a long time. You can limit the search to certain packages by using the optional arguments to who-calls. **Apropos** takes one argument, a string, and finds all symbols whose print-names include the string as a substring. This also takes a long time, and also can be limited to certain packages by using the optional arguments. **Disassemble** takes one argument, the name of a compiled function or the function object itself. It prints a human-readable version of the compiled instructions. **Grindef** pretty-prints the definition of a non-compiled (interpreted) function.

See also the Zmacs Meta-X commands List Callers, Edit Callers, Function Apropos, List Combined Methods, Edit Combined Methods, List Methods, and Edit

Methods.

Section 17.1.1 of volume 1 also describes some variables whose values are automatically maintained by the lisp command loop. *, for instance, is always bound to the value returned by the last form you typed in, and can be extraordinarily helpful.

3.5 The Input Editor and Histories

The *input editor* is active in most contexts outside of the editor. Most notably, it is active when you're typing to a lisp listener. c-Help lists all the input editor commands. Most of them are similar to the Zmacs commands, so you can do all sorts of editing of the input before it gets to the lisp reader. Two of the more helpful features of the input editor are the *histories* it keeps, the *input history* and the *kill history*. Every time you send a form off to be evaluated by a lisp listener, the form is added to that lisp listener's input history. (Each lisp listener keeps its own input history, even the editor's typeout window.) Pressing the Escape key will display the input history of the window you are typing to. Every time you delete more than one character of text with a single command (with, for example, m-D, m-Rubout, Clear Input, c-W, c-K), the deleted text is added to the kill history. There is only one kill history; it is shared by all the windows which use the input editor, and also the Zmacs window(s). c-Escape displays the kill history.

In both the input editor and in Zmacs, c-Y *yanks* the most recent item off the kill history and inserts it at the current cursor position. You can select an earlier element from the history by giving c-Y a numeric argument. Typing m-Y immediately after a c-Y does a *yank pop*; it replaces the text which has just been yanked with the previous element from the history. Repeatedly typing m-Y will rotate all the way through the history. Giving m-Y a numeric argument will jump that many items in the history.

Note that since all windows share the same kill history, it provides a simple way of transferring text from the editor into a lisp listener: just push the text onto the kill history while in the editor, perhaps with c-W or m-W or a mouse command. Then switch to a lisp listener, type c-Y, and presto! There's your text.

In the input editor, c-m-Y yanks from the input history. (c-C also has this effect. It is an older command which some people still prefer to use. I find c-m-Y easier to remember.) m-Y again has the effect of rotating through the history. (m-C is the corresponding older command — you used to need different commands for rotating through the kill history and the input histories, but now m-Y does both.)

In Zmacs, c-m-Y has the effect of yanking from what's called the command history, which is a history of all editing commands which use the mini-buffer. Immediately after a c-m-Y, m-Y has the usual effect.

The ability to yank previous inputs into a lisp listener raises an interesting question: how does the input editor know when you're finished editing and ready for the input to be sent off to lisp? Normally, if you just type your input without any yanking, the input editor knows you're done when you type some sort of delimiter at the end of the input string, like a close paren, to complete a well-formed lisp expression. But if you've yanked an already well-formed expression, how can you complete it? The answer is that there is a special *activation character*. It is the End key. Pressing End while anywhere within a well-formed expression tells the input editor you're done, and it sends your input off to lisp. So if you've yanked a previous input with c-m-Y, you can type End immediately to re-evaluate the same expression, or you can edit it some and type End when finished, to evaluate the modified expression.

There's one other input editor command of special interest. That's c-sh-A, which displays the argument list of the function whose name you have typed. So if I type "`(with-open-file `" to a lisp listener, and then type c-sh-A, the following will appear on my screen:

```
WITH-OPEN-FILE (MACRO): ((STREAM-VARIABLE FILENAME . OPTIONS) &BODY BODY)
```

c-sh-A also works in Zmacs.

3.6 Fun and Games

More definitions from *The Hacker's Dictionary* (Guy L. Steele Jr., *et al*), prompted by my spontaneous use of the term *loser*.

LOSE *verb*.

1. To fail. A program loses when it encounters an exceptional condition or fails to work in the expected manner.
2. To be exceptionally unaesthetic.
3. Of people, to be obnoxious or unusually stupid (as opposed to ignorant). See LOSER.

DESERVE TO LOSE *verb*. Said of someone who willfully does THE WRONG THING, or uses a feature known to be MARGINAL. What is meant is that one deserves the consequences of one's losing actions. "Boy, anyone who tries to use UNIX deserves to lose!"

LOSE, LOSE *interjection.* A reply or comment on an undesirable situation. Example: "I accidentally deleted all my files!" "Lose, lose."

LOSER *noun.* An unexpectedly bad situation, program, programmer, or person. Someone who habitually loses (even winners can lose occasionally). Someone who knows not and knows not that he knows not. Emphatic forms are "real loser," "total loser," and "complete loser."

LOSS *noun.* Something (but not a person) that loses: a situation in which something is losing.

WHAT A LOSS! *interjection.* A remark to the effect that a situation is bad. Example: Suppose someone said, "Fred decided to write his program in ADA instead of LISP." The reply "What a loss!" comments that the choice was bad, or that it will result in an undesirable situation — but may also implicitly recognize that Fred was forced to make that decision because of outside influences. On the other hand, the reply "What a loser!" is a more general remark about Fred himself, and implies that bad consequences will be entirely his fault.

LOSSAGE *(lawss':j) noun.* The stuff of which losses are made. This is a collective noun. "What a loss!" and "What lossage!" are nearly synonymous remarks.

WIN

1. *verb.* To succeed. A program wins if no unexpected conditions arise. Antonym: LOSE.
2. *noun.* Success, or a specific instance thereof. A pleasing outcome. A FEATURE. Emphatic forms: MOBY win, super-win, hyper-win. For some reason "suitable win" is also common at MIT, usually in reference to a satisfactory solution to a problem. Antonym: LOSS.

BIG WIN *noun.* The results of serendipity.

WIN BIG *verb.* To experience serendipity. "I went shopping and won big; there was a two-for-one sale."

WINNER *noun.* An unexpectedly good situation, program, programmer, or person. Albert Einstein was a winner. Antonym: LOSER.

REAL WINNER *noun.* This term is often used sarcastically, but is also used as high praise.

WINNAGE *(win':j) noun.* The situation when a LOSSAGE is corrected or when something is winning. Quite rare. Usage: also quite rare.

WINNITUDE *(win':-tood) noun.* The quality of winning (as opposed to WINNAGE, which is the result of winning).

3.7 Problem Set #3

Questions

Evaluate `(make-system 'funny-window :noconfirm :silent)`. This will load some code, and create and select a window of flavor **funny-window**, making it the value of the symbol ***funny-window***.

The funny-window flavor is built on **tv:lisp-listener**, so you can type forms to this window and it will evaluate them just as a lisp listener would.

This window has two behavioral quirks, one which occurs just before it types out the return value of whatever it evaluates, and one which occurs whenever you move the mouse.

The assignment is to find out what's responsible for the odd behavior. You win the game if you can get the code causing the behavior into a Zmacs buffer. Use whatever you know about looking into the state of the lisp machine. Several good techniques are buried in the text of this chapter.

There's one restriction on what you may do: you should read the code into the editor by using some system-provided operation which finds the definition of a given function. So you have to first figure out what function is responsible. It's cheating to randomly read in files and scan them.

I would suggest starting on the typeout quirk — it should be a little easier to track down than the mouse behavior.

Hints

Note that there's a delay partway through the funny typeout. To find out what's going on, you can simply do c-m-Suspend during the delay and look at what's on the stack. (At this point you might want to `(setq *funny-typeout-delay* 0)` for the duration.) Use c-E on the appropriate frame to pull the offender into the editor.

Finding the mouse funniness isn't as easy. c-m-Suspend doesn't work — and not only because there's no time for it. Try setting ***funny-mouse-delay*** to 1 (sec). Now there's time to type c-m-Suspend during the funniness, but the stack shows nothing revealing, because the action is in a different process. (When convinced, you should probably set *funny-mouse-delay* back to 0.)

One possible way to proceed would be to put a trace (with :error) on the `:draw-circle` method, which you can see is being called. [To get the method do `(get-handler-for *funny-window* :draw-circle)`, which you will see turns out to be `(:method tv:graphics-mixin :draw-circle)`]. That would force the process which is calling `:draw-circle` into the debugger. Unfortunately everything gets fouled up when the mouse process, which turns out to be the culprit, goes into the debugger. (The trouble starts because the mouse process doesn't have a window to use, so it has to come up with one and expose it. That in itself is okay [processes which run in the background generally have that capability], but the real killer is that the mouse process has to be running in order for most ways of switching windows to work. With the mouse process halted for debugging, most window system operations are impossible, and the lisp machine is suddenly in a terribly wedged* state! (It usually isn't necessary to re-boot if this happens — try resetting the mouse process, with `(send tv:mouse-process`

* **WEDGED** *adjective.*

1. To be stuck, incapable of proceeding without help. This is different from having CRASHED. If the system has crashed, then it has become totally nonfunctioning. If the system is "wedged," it is trying to do something but cannot make progress. It may be capable of doing a few things, but not be fully operational. For example, the system may become wedged if the disk controller FRIES; there are some things you can do without using the disks, but not many. Being wedged is slightly milder then being "hung." This term is sometimes used as a synonym for DEADLOCKED. See also HANG, LOSING, CATATONIA, and BUZZ.
2. Of a person, suffering severely from misconceptions. Examples: "He's totally wedged — he's convinced that he can levitate through meditation." "I'm sorry. I had a BIT set that you were responsible for TECO, but I was wedged."

(*The Hacker's Dictionary*, Guy L. Steele, Jr., *et al*)

`:reset`). If you can't get to a window that will accept typein, do Function-Suspend first.)

If you really want to make this approach work, here's something a little more advanced (that's understated — it took two of us quite a while to come upon it) that will produce the desired effect:

```
(advise (:method tv:graphics-mixin :draw-circle) :before nil nil
  (send (send *funny-window* :process)
        :interrupt #'dbg current-process)
  (process-sleep 60.))
```

The basic idea is to force the process calling `:draw-circle` into the debugger by calling the **dbg** function on it, but to do the call to the dbg function from the funny window, using its process and its already exposed and selected window. Now you can examine the stack and quickly find the `:before :mouse-moves` method which is responsible.

But there's a much more sensible approach. You could think of the above force-it-into-the-debugger approach as finding the set of functions which are being called and filtering that set for functions which seem related to the funny-window flavor. Turning that around is far more effective in this case. That is, first collect the functions related to the funny-window flavor, which is much easier in this case than getting access to the control stack, and then filter that set for ones likely to be causing the funny effects. So one idea is to do a `:which-operations` on the window and search for messages with suggestive names. You could, for instance, try

```
(loop for msg in (send *funny-window* :which-operations)
      when (string-search "mouse" msg) collect msg)
```

[If you have the Murray Hill standard utilities, you could simply do `(grep-msgs "mouse" *funny-window*)`.] This will be a small enough set that they could all be investigated. A more direct approach, and the clear winner for this problem, is based on the observation that although the funny-window flavor has many methods defined, the bulk of them are inherited from other flavors, and whatever is responsible for its peculiar behavior must be somewhere in the relatively small number of methods defined locally for funny-window. So all you have to do is enter the Flavor Examiner (Select-X), and get the list of local methods for funny-window. How about that. Two methods, one for each peculiarity. Now you can edit them either by clicking mouse-right while pointing to one, and choosing "edit" from the menu, or by clicking where it says "edit" on the extreme right.

Solutions

Here's what causes the typeout funniness:

```
(defvar *funny-typeout-delay* 1)

;;; You found the funny typeout stuff!
(defwhopper (funny-window :input-editor) (&rest args)
  (let ((thing (lexpr-continue-whopper args)))
    (format self "~&You entered something of type ")
    (process-sleep (* 60 *funny-typeout-delay*))
    (princ (typep thing) self)
    thing))
```

And this causes the mouse funniness:

```
(defvar *funny-mouse-delay* 0)

;;; You found the funny mouse stuff!
(defmethod (funny-window :before :mouse-moves) (x y)
  (format self "~&The mouse is moving . . . ")
  (process-sleep (* 60 *funny-mouse-delay*))
  (send self :draw-circle
        (- x tv:left-margin-size) (- y tv:top-margin-size) 8)
  (send self :tyo #\!))
```

Chapter 4

FLOW OF CONTROL

In this chapter we leave behind the "operating system" of the lisp machine and return to aspects of the lisp language itself. In particular, we'll look at the various constructs for determining the flow of control. The notes are a little sketchier than usual, because this material is covered reasonably well in Part V of volume 2 of the Symbolics documentation.

4.1 Conditionals

The **cond** special form is the basic conditional. The arguments to **cond** are any number of *clauses*, where each clause is a list containing a predicate (the *antecedent*) and zero or more *consequents*. The clauses are handled one at a time, in order of appearance. If a clause's antecedent returns nil, its consequents are skipped and the next clause is considered. When a clause is found whose antecedent returns a non-nil value, that clause's consequents, if any, are all evaluated. The value of the last consequent in that clause (or of the antecedent, if there are no consequents) is the value returned by the **cond**, and the rest of the clauses are skipped. If every clause's antecedent returns nil, the **cond** returns nil.

Beyond their value as logical operators, **and** and **or** can also be used as conditionals. If one of the subforms of an **and** returns nil, the rest will be skipped. And if one of the subforms of an **or** returns non-nil, the rest will be skipped. Here are two examples taken from the manual:

```
(and bright-day
     glorious-day
     (print "It is a bright and glorious day."))

(or it-is-fish
    it-is-fowl
    (print "It is neither fish nor fowl."))
```

Almost all of the remaining conditionals are really macros which expand into calls to cond. **when** and **unless** take a predicate and any number of consequent forms. If the predicate for a **when** returns non-nil, all of the consequent forms are evaluated and the value returned by the last one is returned by the **when**. Otherwise (a nil predicate value) the consequent forms are all skipped. **unless** works similarly: if the predicate returns nil the consequents are evaluated, otherwise they're skipped. Here are the definitions of the **when** and **unless** macros:

```
(DEFMACRO WHEN (TEST &BODY BODY)
  `(COND (,TEST (PROGN ,@BODY))))

(DEFMACRO UNLESS (TEST &BODY BODY)
  `(COND ((NOT ,TEST) (PROGN ,@BODY))))
```

[Actually, I lied a bit in the previous paragraph. Although they could, and in fact used to, have the given macro definitions, **when** and **unless** are now *special forms* instead. I used the old macro definitions because they're easier to read, while for our immediate purposes, any differences in behavior aren't important.]

Defining suitable macros is an extremely common (and effective) method for extending the syntax of lisp. There are no extra levels of function calling at run-time, and no modifications to the compiler are involved.

if is somewhat similar to **when**, and also macroexpands to a call to **cond**. If the test evaluates to non-nil, the first body form is evaluated. Otherwise (test returns nil), if there are any body forms following the first, they are all evaluated.

Three more macros based on **cond** are **select**, **selector**, and **selectq**. **Selectq** is used most frequently. Here is its structure:

```
(selectq key-form
   (test consequent consequent ...)
   (test consequent consequent ...)
   (test consequent consequent ...)
   ...)
```

The key-form is evaluated and checked against the tests. The tests are not evaluated. Each test must be either a symbol, a fixnum, or a list of symbols and fixnums. If the test (or one of its elements, if it's a list) is **eq** to the evaluated key-form, then it matches. The symbols **t** and **otherwise** are special: a test consisting of one of those two symbols always matches. As with **cond**, the consequents to the first test which matches are evaluated, and the rest of the clauses are skipped.

Select is the same as **selectq**, except that the symbols in the tests are evaluated before being compared to the (evaluated) key-form. **Selector** is like **select** except that there is an additional argument (following the key-form) which is the name of a function to use for the comparison in place of eq.

See also **typecase**, **dispatch**, **cond-every**, and **selectq-every**.

4.2 Blocks and Exits

block is the primitive for defining a piece of code which may be exited from the middle. The first argument to **block** must be a symbol. It is not evaluated, and becomes the name of the block. The rest of the arguments are forms to be evaluated. If a call to **return-from** occurs within the block, with a first argument of the block's name, the block is immediately exited. The rest of the arguments to **return-from** are evaluated and become the return values for the block. (Beginning with release 6.0, the preferred way of returning multiple values is with **(return-from** *name* **(values** *form...***))**, for compatibility with Common Lisp.)

The scope of the block is lexical, so the corresponding call to return-from must occur textually within the block; it will not work to call return-from inside a function which is called within the block. The next section discusses that sort of nonlocal exit.

Blocks may be nested; that's the whole point of naming them. A return-from causes an immediate exit from the innermost block with a matching name.

Some other constructs (including **do** and **prog**) create implicit blocks. These blocks have **nil** for a name, and so they may be prematurely exited with `(return-from nil (values` *value...*`))`. They may also be exited with the **return** special form, which always exits the innermost block named **nil**.

4.3 Nonlocal Exits

catch and **throw** are analogous to **block** and **return-from**, but are scoped dynamically rather than lexically. This means that a **throw** may cause an exit from any **catch** on the control stack at the time the **throw** is reached (unless an inner **catch** is shadowing an outer **catch** with the same tag). **catch**'s equivalent to the name of a block is its *tag*. The tag is the first argument to **catch**; it is evaluated, and may return any lisp object. The first argument to **throw** is its tag. It is also evaluated, and the **throw** causes the exit of the innermost **catch** whose evaluated tag is **eq** to the **throw**'s evaluated tag.

If a **throw** occurs, its second argument is evaluated and the value(s) returned will be returned by the corresponding **catch**. If no **throw** occurs, the values returned by the **catch** are the values returned by its last subform.

There are obsolete forms of **catch** and **throw**, called ***catch** and ***throw**. They differ from the newer version mainly in what values are returned. ***catch** and ***throw** should not be used in new code.

4.4 Iteration

There are three styles of built-in facilities for iteration. A group of operators are available for mapping a function through one or more lists; the **do** special form allows more general forms of iteration; and the **loop** macro provides even more flexibility. (I should also point out that functions which make use of tail recursion are compiled into iterative structures.) This set, of course, could easily be extended by writing more macros.

When more than one of the iteration facilities is applicable to a particular task, the choice is mainly a matter of personal taste.* All three are comparably efficient. The key issue is readability, and on this score opinions differ. My own view is that the mapping functions are succinct and to the point, and therefore desirable, within the limited set of applications that are easily expressed as mapping operations. For all other kinds of iteration, **do** is often concise but I find it somewhat obscure.† I

* See hacker's definition at end of chapter.

think **loop** is much easier to read, but there are those who consider it wordy and nebulous.

> "**Loop** forms are intended to look like stylized English rather than Lisp code. There is a notably low density of parentheses, and many of the keywords are accepted in several synonymous forms to allow writing of more euphonious and grammatical English. Some find this notation verbose and distasteful, while others find it flexible and convenient. The former are invited to stick to **do**."
>
> [The preceding, as well as parts of the discussion of **loop** below, is taken from the loop documentation written by Glenn S. Burke: MIT Laboratory for Computer Science TM-169.]

For the sake of comparison, here are four ways to print the elements of a list:

```
(defun print-elts1 (list)              ; mapc
  (mapc #'print list))

(defun print-elts2 (list)              ; do
  (do ((l list (cdr l)))
      ((null l))
      (print (car l))))

(defun print-elts3 (list)              ; loop
  (loop for elt in list
        do (print elt)))

(defun print-elts4 (list)              ; tail recursion
  (unless (null list)
    (print (car list))
    (print-elts4 (cdr list))))
```

4.4.1 Mapping

The mapping operators come in six varieties. They all take as arguments a function followed by any number of lists, and step through all the lists in parallel, applying the function as they go. They vary along two dimensions: whether they operate on successive elements of the lists or on successive sublists, and which of

† ditto.

three kinds of values they return. (Two x three = six.) The return value can be simply the second argument to the mapping operator (implying that the mapping was done for side effect only), or a list of the results of function application, or a list formed by nconcing (destructive appending) the results of function application. Here are a few examples:

Applied to successive elements, lists results

```
  (mapcar #'+ '(1 2 3 4) '(2 4 6 8))
→ (3 6 9 12)

  (mapcar #'list '(1 2 3 4) '(2 4 6 8))
→ ((1 2) (2 4) (3 6) (4 8))
```

Successive elements, nconcs results

```
  (mapcan #'list '(1 2 3 4) '(2 4 6 8))
→ (1 2 2 4 3 6 4 8)
```

Successive sublists, lists results

```
  (maplist #'(lambda (l) (and (second l)
                              (+ (first l) (second l))))
           '(1 2 3 4))
→ (3 5 7 NIL)

  (maplist #'(lambda (l) (list (first l) (second l)))
           '(1 2 3 4))
→ ((1 2) (2 3) (3 4) (4 NIL))
```

Successive sublists, nconcs results

```
  (mapcon #'(lambda (l) (list (first l) (second l)))
          '(1 2 3 4))
→ (1 2 2 3 3 4 4 NIL)
```

4.4.2 Do

A **do** looks like this:

(do ((*var init repeat*)

```
      (var init repeat) ...)
     (end-test exit-form exit-form ...)
    body-form body-form ...)
```

The first subform is a list of index variable specifiers. Upon entering the **do**, each *var* is bound to the corresponding *init*. And before each subsequent iteration, *var* is set to *repeat*. The variables are all changed in parallel. The *end-test* is evaluated at the beginning of each iteration. If it returns a non-nil value, the *exit-forms* are all evaluated, and the value of the last one is returned as the value of the **do**. Otherwise the *body-forms* are all evaluated. Here's an example which fills an array with zeroes:

```
(do ((i 0 (1+ i))
     (n (array-length foo-array)))
    ((= i n))
  (setf (aref foo-array i) 0))
```

Upon entry, i is bound to 0 and n is bound to the size of the array. On each iteration, i is incremented. (n stays constant because it has no *repeat* form.) When i reaches n, the **do** is exited. On each iteration, the *i*th element of foo-array is set to 0.

And another, which is equivalent to `(maplist #'f x y)`:

```
(do ((x x (cdr x))
     (y y (cdr y))
     (z nil (cons (f x y) z)))
    ((or (null x) (null y))
     (nreverse z)))
```

Note that the preceding example has no body. It's actually fairly common for all the action in a **do** to be in the variable stepping.

There are macros named **dotimes** and **dolist** which expand into common **do** constructs. For instance, the following macroexpands into the first example above:

```
(dotimes (i (array-length foo-array))
  (setf (aref foo-array i) 0))
```

4.4.3 Loop

loop is a macro which expands into a **prog** with explicit **go** statements, and often with several lambda bindings. A typical call looks like this:

```
(loop clause
      clause
      clause ...)
```

Each clause begins with a keyword, and the contents of the rest of the clause depend on which keyword it is. Some clauses specify variable bindings and how the variables should be stepped on each iteration. Some specify actions to be taken on each iteration. Some specify exit conditions. Some control the accumulation of return values. Some are conditionals which affect other clauses. And so on. A full discussion of all the clauses would be lengthy and not particularly useful, as they're all described coherently enough in the documentation. We'll just look at some representative examples.

The **repeat** clause specifies how many times the iteration should occur. The keyword is followed by a single lisp expression, which should evaluate to a fixnum. And the **do** keyword is followed by any number of lisp expressions, all of which are evaluated on each iteration. So putting the two together,

```
(loop repeat 5
      do (print "hi there"))
```

prints "hi there" five times. The most commonly used (and complicated) of the iteration-driving clauses is the **for** clause. The keyword is followed by the name of a variable which is to be stepped on each iteration, then some other stuff which somehow specifies the initial and subsequent values of the variable. Here are some examples:

```
(loop for elt in expr
      do (print elt))
```

expr is evaluated (it better return a list), elt is bound in turn to each element of the list, and then the loop is exited

```
(loop for elt on expr
      do (print elt))
```

similar, but elt is bound to each sublist

```
 ... for x = expr ...
```

expr is re-evaluated on each iteration and x is bound to the result (no exit specified here)

```
 ... for x = expr1 then expr2 ...
```

x is bound to *expr1* on the first iteration and *expr2* on all succeeding iterations

`... for x from` *expr* `...`	x is bound to *expr* (it better return a fixnum) on the first iteration and incremented on each succeeding iteration
`... for x from` *expr1* `to` *expr2* `...`	like above, but the loop is exited after x reaches *expr2*
`... for x from` *expr1* `below` *expr2* `...`	like above, but the loop is exited just before x reaches *expr2*
`... for x from` *expr1* `to` *expr2* `by` *expr3* `...`	incremented by *expr3* on each iteration
`... for x being` *path* `...`	user definable iteration paths

When there are multiple **for** clauses, the variable assignments occur sequentially by default (parallel assignment may be specified), so one **for** clause may make use of variables bound in previous ones:

```
(loop for i below 10          i starts at 0 when from isn't specified
      for j = (* i i) ...)
```

The **with** clause allows you to establish temporary local variable bindings, much like the **let** special form. It's used like this:

```
(loop with foo = expr   expr is evaluated only once, upon entering the loop
      ... )
```

A number of clauses have the effect of accumulating some sort of return value. The form

```
(loop for item in some-list
      collect (foo item))
```

applies foo to each element of some-list, and returns a list of all the results, just like `(mapcar #'foo some-list)`. The keywords **nconc** and **append** are similar, but the results are nconc-ed or appended together. Keywords for accumulating numerical results are **count**, **sum**, **maximize**, and **minimize**. All of these clauses may optionally specify a variable into which the values should be accumulated, so that it may be referenced. For instance,

```
(loop for x in list-of-frobs
      count t into count-var        "t" means always count
      sum x into sum-var
      finally (return (// sum-var count-var)))
```

computes the average of the entries in the list.

The **while** and **until** clauses specify explicit end-tests for terminating the loop (beyond those which may be implicit in **for** clauses). Either is followed by an arbitrary expression which is evaluated on each iteration. The loop is exited immediately if the expression returns the appropriate value (nil for **while**, non-nil for **until**).

```
(loop for char = (send standard-input :tyi)
      until (char-equal char #\end)
      do (process-char char))
```

The **when** and **unless** clauses conditionalize the execution of the following clause, which will often be a **do** clause or one of the value accumulating clauses. Multiple clauses may be conditionalized together with the **and** keyword, and *if-then-else* constructs may be created with the **else** keyword.

```
(loop for i from a to b
      when (oddp i)
        collect i into odd-numbers and do (print i)
        else collect i into even-numbers)
```

The **return** clause causes immediate termination of the loop, with a return value as specified:

```
(loop for char = (send standard-input :tyi)
      when (char-equal char #\end)
        return "end of input"
      do (process-char char))
```

Please refer to the documentation for a more complete discussion of **loop** features. Of particular importance are *prologue* and *epilogue* code (the **initially** and **finally** keywords), the distinction between ways of terminating the loop which execute the epilogue code and those which skip it, aggregated boolean tests (the **always**, **never**, and **thereis** keywords), the destructuring facility, and iteration paths (user-definable iteration-driving clauses).

4.5 A Bit More on Working with Macros

Since so many of the control structure constructs are macros, you'll probably find it helpful in working with them to be able to see what forms they macroexpand into. The most basic of the tools for expanding macros is the **macroexpand** function. It takes a list as an argument and returns its expanded version:

```
  (macroexpand '(if (test) (do-this) (do-that))))
→ (COND ((TEST) (DO-THIS)) (T (DO-THAT)))
```

The **grind-top-level** function, for pretty-printing, is also useful in conjunction with **macroexpand**.

Rather than type `(grind-top-level (macroexpand ... ))` repeatedly, an easier way to expand forms in a lisp listener is with **mexp**. Evaluating `(mexp)` puts you into a macro reading and expanding loop. You type a form, and it pretty-prints the expansion and waits for another input form. Exit by pressing the *End* key.

But by far the easiest way to see what something expands into is the c-sh-M command in the editor. When you place the cursor at the beginning of a lisp expression and type c-sh-M, the macro-expansion of the form is shown in the typeout window. If you give c-sh-M a numeric argument, it inserts the expanded form into the buffer. c-sh-M only expands the top-level form, not any of its subforms which are themselves macros. m-sh-M expands subforms as well.

Of course, you can also use Meta-. on a macro to edit its definition, but it's often more useful to see what a macro expands into than to see how it does it.

4.6 Fun and Games

From *The Hacker's Dictionary*, Guy L. Steele, Jr., *et al*:

TASTE *noun.*
  Aesthetic pleasance; the quality in programs which tends to be inversely proportional to the number of FEATURES, HACKS, CROCKS, and KLUGES programmed into it.

OBSCURE *adjective.*
  Little-known; incomprehensible; undocumented. This word is used, in an exaggeration of its normal meaning, to imply a total lack of

comprehensibility. “The reason for that last CRASH is obscure.” “That program has a very obscure command syntax.” “This KLUDGE works by taking advantage of an obscure FEATURE in TECO.” The phrase “moderately obscure” implies that it could be figured out but probably isn’t worth the trouble.

4.7 Problem Set #4

Questions

1. A. Write a function called **do-word** which takes one required argument, a symbol, and two optional arguments, both fixnums. If the symbol is one of `:point`, `:rectangle`, `:triangle`, or `:circle`, the function should draw the specified kind of object on standard-output, at the coordinates specified by the fixnum arguments (choose any size you like for the figures). If the symbol is `:reverse`, the function should toggle the state of `reverse-video-p` for standard-output. If the symbol is `:erase`, the function should clear standard-output. If the symbol is anything else, it should call the function **beep**.

   B. Write another function, called **read-chars**, which reads a series of characters from the keyboard. When the Return key is pressed, it makes a string out of all the characters read (before Return), and returns a symbol in the keyword package whose print-name is that string. That is, if the characters "s," "a," "m," "p," "l," "e" and "<cr>" were typed, the symbol `:sample` would be generated. (Normally, you wouldn't write such a function because there are already several built-in functions which do the same sort of thing and with more features. But for the sake of demonstrating a point or two...)

   C. Write a third function, called **main-loop**, which repeatedly calls **read-chars** and then **do-word** on the result. It should provide the optional arguments to **do-word**, and increment one or both each time so that the figures are drawn across the top of the screen, and then in a second row below the first, and so on, with each position 200 pixels from the previous one. That is, if standard-output is a window whose dimensions are 1088 by 749, the optional arguments should be (0,0), then (200,0), (400,0), (600,0), (800,0), (0,200), (200,200) ... (400,800). (Don't worry about resetting the coordinates after an `:erase`, or not stepping after a `:reverse` — just allow blank spots.) This version of **main-loop** should iterate until it reaches the bottom right corner of the window. You'll probably want to touch up **do-word** so that it draws the figures centered in the 200 x 200 area whose upper left corner it is given.

   D. Alter **main-loop** and (read-chars) so that if at any point you press the End key, even in the middle of typing a word, **main-loop** will return immediately.

   E. Make similar alterations so that the Line key will have the effect of skipping any remaining positions in the current row and continuing at the beginning of the next row.

2. Write three functions which meet the following spec, one using a mapping operator, one using **do**, and one using **loop**: **f** takes one argument, a list of fixnums, and returns a list containing only the even numbers from its argument. That is,
   `(evens '(1 2 3 4 5))  →  (2 4)`

3. Like (2), but this time the list which **f** returns includes only those elements of the argument which are less than the succeeding element. That is,
   `(less-thans '(3 6 7 2 5 4 3 5))  →  (3 6 2 3)`

4. Write a function which takes two arguments, a fixnum "n" and a list "l," and uses a mapping operator to return a list consisting of the elements of l incremented by n. Try to do this two ways, one using a lambda expression as the function argument to the mapping operator, and one just using `#'+`.

5. Write a function which takes one argument, a fixnum n, and returns the nth fibonacci number, subject to the restriction that the time it takes should increase linearly with n. I suggest using **loop**.

6. Write a function of no arguments which finds all prime numbers between 2 and 50000. (You might find it easier to have the function stop at the first prime greater than 50000.) I suggest using **loop** again.

7. The function **mexp** only expands the initial operator in an expression. If one of the arguments to that operator itself uses a macro, that doesn't get expanded. That is,

```
(if test (do-this)
    (if another-test (do-that)
        (do-the-other)))
```

   expands into

```
(COND (TEST
       (DO-THIS))
      (T (IF ANOTHER-TEST (DO-THAT) (DO-THE-OTHER))))
```

   instead of

```
(COND (TEST
       (DO-THIS))
      (T (COND (ANOTHER-TEST
               (DO-THAT))
              (T (DO-THE-OTHER)))))
```

Write **mexp-all**, which behaves just like **mexp** except that it macroexpands subexpressions and thus would produce the second of the expansions shown above.

Hints

1. A. The body of **do-word** should consist of a **selectq** (or a **defselect**, which is a macro that expands into a **selectq**). The messages you'll need to send standard-output are `:draw-point`, `:draw-rectangle`, `:draw-triangle`, `:draw-circle`, `:reverse-video-p`, `:set-reverse-video-p`, and `:clear-window`.

   B. To read each character, send `:tyi` to standard-input or terminal-io (usually the same thing). Collect the characters into a list, and use **string-append** to combine the characters into a string, and **intern** to make the string into a symbol. Make sure the characters are converted to upper case, either before or after they're collected into the string.

   C. The `:inside-size` message to a window returns two values, the width and height of the interior, in pixels. (The difference between inside-size and size is the margin area, which the usual output messages can't draw in.) Use these (actually, 200 less than these) as the upper bounds on your iteration. You probably want two nested loops, an outer one for the y coordinate and an inner one for the x coordinate.

   D. Wrap the body of **main-loop** in a **catch**, and have **read-chars** do a **throw** at the appropriate point.

   E. Same technique. This time the **catch** goes around the inner loop.

2. The mapping operator should be **mapcan**. The **do** would need a structure very much like that produced by the **dolist** macro (which you might even want to use), combined with some parts of the **do** example in the notes which was equivalent to **maplist**. The **loop** would use `collect`, conditionalized with `when`.

3. This time the mapping operator should be **mapcon**. The **do** and **loop** can be done in any of several ways. There could be one iteration variable which took on the value of successive sublists, and then you could take its first and second elements and do the comparison. But better (because it requires fewer redundant cdr's and exits more cleanly) would be to have the value of the "first" variable be directly set to the previous value of the "second" variable.

4. The lambda expression adds n to its argument. A very nice way to do this without a lambda expression is to use a circular list. (You can make one yourself with **rplacd**, but the **circular-list** function is more convenient.)

5. Lots of choices here. You probably want to use a `repeat` clause to control the loop termination. But the tricky part is that you need to have a value

from one iteration hang around through the next one. So you might want something like the loop answer to (3), where one variable always takes on the previous value of another one. And you'll probably want to use a `finally` clause with an explicit call to **return**.

6. You probably want a subroutine which tests whether a number "n" is factorable by any of a list "l" of potential factors. The loop keyword `thereis` may be helpful in writing this part. Then you'll want to use this in another loop which tests each number from 2 up for factorability. When it finds a prime, it should add it to the list of potential factors.

7. The editor commands Macro Expand Expression (c-sh-M) and Macro Expand Expression All (m-sh-M) display exactly the same difference in behavior we wish to see between **mexp** and **mexp-all**. So you should be able to pull some code out of Macro Expand Expression All, and use it as a black box to help you create **mexp-all**.

Solutions

1. A.
```
(defun do-word (message &optional x y)
  (selectq message
    (:point (send standard-output :draw-point x y))
    (:rectangle (send standard-output :draw-rectangle
                        100 100 x y))
    (:triangle (send standard-output :draw-triangle
                       x y (+ x 100) y (+ x 50) (- y 87)))
    (:circle (send standard-output :draw-circle x y 50))
    (:reverse (send standard-output :set-reverse-video-p
                      (not (send standard-output
                                 :reverse-video-p))))
    (:erase (send standard-output :clear-window))
    (otherwise (beep))))
```

B.
```
(defun read-chars ()
  (loop for char = (send terminal-io :tyi)
        until (char-equal char #\return)
        collect (char-upcase char) into char-list
        finally (return (intern (apply #'string-append
                                       char-list)
                                "keyword"))))
```

C.
```
(defun main-loop ()
  (multiple-value-bind (width height)
      (send standard-output :inside-size)
    (loop for y below (- height 200) by 200
          do (loop for x below (- width 200) by 200
                   do (do-word (read-chars) x y)))))
```

and in do-word

```
(:point (send standard-output :draw-point
              (+ x 100) (+ y 100)))
(:rectangle (send standard-output :draw-rectangle
                  100 100 (+ x 50) (+ y 50)))
(:triangle (send standard-output :draw-triangle
                 (+ x 100) (+ y 57) (+ x 50) (+ y 143)
                 (+ x 150) (+ y 143)))
(:circle (send standard-output :draw-circle
               (+ x 100) (+ y 100) 50))
```

D.
```
(defun main-loop ()
  (catch 'end-key
    (multiple-value-bind (width height)
        (send standard-output :inside-size)
      (loop for y below (- height 200) by 200
            do (loop for x below (- width 200) by 200
                     do (do-word (read-chars) x y))))))

(defun read-chars ()
  (loop for char = (send terminal-io :tyi)
        until (char-equal char #\return)
        when (char-equal char #\end)
          do (throw 'end-key nil)
        collect (char-upcase char) into char-list
        finally (return (intern (apply #'string-append
                                       char-list)
                                "keyword"))))
```

E.
```
(defun main-loop ()
  (catch 'end-key
    (multiple-value-bind (width height)
        (send standard-output :inside-size)
      (loop for y below (- height 200) by 200
            do (catch 'line-key
                 (loop for x below (- width 200) by 200
                       do (do-word (read-chars)
                                   x y)))))))

(defun read-chars ()
  (loop for char = (send terminal-io :tyi)
        until (char-equal char #\return)
        when (char-equal char #\end)
          do (throw 'end-key nil)
        when (char-equal char #\line)
          do (throw 'line-key nil)
        collect (char-upcase char) into char-list
        finally (return (intern (apply #'string-append
                                       char-list)
                                "keyword"))))
```

2.
```
(defun evens (list)
  (mapcan #'(lambda (n) (and (evenp n) (list n))) list))
```

```
   (defun evens (list)
     (do ((sublist list (cdr sublist))
          (item)
          (evens-found))
         ((null sublist)
          (nreverse evens-found))
       (setq item (car sublist))
       (if (evenp item) (push item evens-found))))

   (defun evens (list)
     (let (evens-found)
       (dolist (item list)
         (if (evenp item) (push item evens-found)))
       (nreverse evens-found)))

   (defun evens (list)
     (loop for item in list
           when (evenp item) collect item))
```

```
3. (defun less-thans (list)
     (mapcon #'(lambda (l)
                 (and (cdr l)      to avoid doing (< n nil) at the end
                      (< (first l) (second l))
                      (list (first l))))
             list))

   (defun less-thans (list)
     (do ((sublist list (cdr sublist))
          (item) (good-ones))
         ((= 1 (length sublist))
          (nreverse good-ones))
       (setq item (car sublist))
       (if (< item (second sublist))
           (push item good-ones))))

   (defun less-thans (list)
     (loop for a = (car list) then b
           for b in (cdr list)
           when (< a b) collect a))
```

```
4. (defun add-to-list (n list)
     (mapcar #'(lambda (elt) (+ elt n)) list))
```

```
(defun add-to-list (n list)
  (mapcar #'+ (circular-list n) list))
```

5. I've fudged the repeat counts of these three so that they'll agree on which is the nth fibonacci number.

```
(defun linear-fib (n)
  (loop repeat n
        for a = 0 then b
        for b = 1 then partial-sum
        for partial-sum = (+ a b)
        finally (return partial-sum)))

(defun linear-fib (n)
  (loop repeat (+ n 2)
        for a = 0 then b
        for b = 1 then partial-sum
        sum a into partial-sum
        finally (return partial-sum)))

(defun linear-fib (n)
  (loop repeat (1+ n)
        with a = 1
        sum (prog1 a (setq a partial-sum))
          into partial-sum
        finally (return partial-sum)))
```

6. This takes about 10 seconds to find all primes up to 50021 (˜5100 of them).

```
(defsubst factorable? (n factors)
  (loop for f in factors
        while (≤ (* f f) n)
        thereis (zerop (remainder n f))))

(defun primes ()
  (loop for n from 2
        unless (factorable? n found)
          collect n into found
          and when (> n 50000) return (length found))))
```

7. The editor command Meta-X Macro Expand Expression All calls a function named **macro-expand-all** to do the actual expansion. It's kind of messy, but it basically does a recursive tree walk of the input form, expanding each part. Understanding how it works, however, is not really necessary. All we need to

know is what it does; we can treat it as a black box. Here is the original definition of **mexp**, and one for **mexp-all**, which modifies **mexp** to use **macro-expand-all**.

```
(DEFUN MEXP ()
  (LOOP WITH (FORM FLAG)
        DOING (FORMAT T "~2&")
              (MULTIPLE-VALUE (FORM FLAG)
                (PROMPT-AND-READ ':EXPRESSION-OR-END
                                 "Macro form → "))
        UNTIL (EQ FLAG ':END)
        DO (LOOP AS NFORM = FORM
                 DO (SETQ FORM (MACROEXPAND-1 NFORM))
                 UNTIL (EQ FORM NFORM)
                 DO (PRINC " → ")
                    (GRIND-TOP-LEVEL FORM))))

(DEFUN MEXP-all ()
  (LOOP WITH (FORM FLAG)
        DOING (FORMAT T "~2&")
              (MULTIPLE-VALUE (FORM FLAG)
                (PROMPT-AND-READ ':EXPRESSION-OR-END
                                 "Macro form → "))
        UNTIL (EQ FLAG ':END)
        DO (PRINC " → ")
           (GRIND-TOP-LEVEL (zwei:macro-expand-all FORM))))
```

Chapter 5

THE GRAPH EXAMPLE

This chapter, rather than present some abstracted features of the lisp language or of the lisp machine operating environment, will cover a programming example which puts to use many of the features we have previously discussed. The piece of code in question allows one to display and manipulate simple undirected graphs, that is sets of *nodes* connected by *arcs*.

If your site has loaded the tape which accompanies this book, you can load the code by using the CP command `Load System graph` [or evaluating `(make-system 'graph)`]. Once the code has been read, start the program by evaluating `(send (tv:make-window 'graph-frame) :select)`.

Please refer to section 5.5, which contains a picture of the program in operation and a listing of the code. The first three sections will point out and briefly discuss the interesting features of the code. The basic mechanism is contained in the first half of the file. All the graph manipulation functionality is there, provided you're willing to type in awkward forms to a lisp listener. The rest of the file provides a more convenient user interface.

5.1 The Nodes and Arcs

The four **defvar**'s

These four declarations are for global variables that will be needed at various places throughout the code. A **defvar** must precede the first appearance of the variable so that the compiler knows the symbol refers to a *special variable*. Another good reason for putting them at the beginning is so anyone reading the code can quickly find out what hooks are available for modifying the program's behavior.

The **node** defflavor

All five instance variables are *settable* (you may recall that settable implies gettable and initable). The `:required-init-keywords` option specifies that every call to **make-instance** must include initial values for `xpos` and `ypos`.

defun-method

This facility is sort of a cross between **defun** and **defmethod**. It's used for defining functions to be called from inside methods, which need access to the instance variables. For defun-methods to be compiled optimally, they should be defined before they are used.

The `:init` method

The last thing **make-instance** always does is call the flavor's `:init` method, if it has one. Here you can specify operations to be performed on every instance of your flavor, upon being instantiated. I use this one to set the radius of the new node according to the size of its label, and to add the node to the list of all nodes. Note that many of the window mixins already have `:init` methods, so if your flavor is built on some of them you'd better use a before or after daemon rather than a primary method so you don't override the other one.

`ignore` as an argument

Use of `ignore` in a lambda-list for an argument which isn't going to be used saves you from getting a compiler warning about an unused variable.

`:send-if-handles`

This method comes with **si:vanilla-flavor**. The specified message is sent only if the object has a handler for it. This avoids unclaimed message errors in cases when you can't be sure of the exact flavor of the object you're dealing with. The

`:primitive-item` message has to do with mouse-sensitive items. We'll get to that a little later.

map-over-arcs

I wrote this macro to have a handy way to iterate over all the arcs. For each node it runs through all its arcs. If it has already seen that arc it goes on. If it hasn't, it executes the body forms with the specified variable bound to the arc, and marks the arc as visited. Note the use of **gensym** to make sure the mark-var and node-var don't shadow any variable bindings.

5.2 Managing Multiple Windows and Processes

We now have everything needed to make and alter graphs, but it would be very awkward to do it by typing forms like

```
(make-instance 'arc
               :node1 (make-instance 'node :xpos 135 :ypos 251)
               :node2 (make-instance 'node :xpos 338 :ypos 92))
```

The rest of the file is concerned with making it easy to do this sort of thing. First we define a *frame* (an object composed of several windows). Our frame has two *panes*, one for display of the graph, and one for typing lisp forms. Then we set up a process in the graph pane which just watches for mouse clicks on the mouse-sensitive items (the nodes) and dispatches appropriately. The lisp pane, by virtue of being built on **tv:lisp-listener**, will automatically have its own process for reading lisp forms, evaluating them, and printing the results.

Managing multiple processes is something people often find confusing, at least the first few times. Windows, processes, and i/o buffers are all independent objects on the lisp machine, so you can have nearly any number of each. A window can have zero or one processes running in it, and any number of processes can exist independently of windows; windows can each have their own i/o buffers, or any number of them can share a single i/o buffer; any process can read from (or stuff characters into) any i/o buffer. How do you decide how many you want? I'm not sure how generally applicable they are, but here are some guidelines you can try out.

First decide on the windows: the number of windows is your number of distinct output channels to the user, so you'll need as many as you have things you want to display simultaneously. In this example we want to see the graph and our interaction with a lisp listener, thus we have two windows.

Next think about your input channels from the user. You'll probably want to use one i/o buffer for each. In our case we need one for keyboard typein (lisp forms) and one for mouse clicks on the mouse-sensitive items. Hopefully it will be obvious how the i/o buffers and windows match up. For this example it is: we'll use the i/o buffer of the window displaying the graph for mouse clicks, and the i/o buffer of the window displaying lisp interaction for keyboard typein. (See chapter 7 for a situation that calls for having two windows share an i/o buffer so that the process running in one can see input from both.)

Now processes. For each i/o buffer there will have to be a process, generally one running in the window to which the i/o buffer belongs. (Having several processes read from a single i/o buffer would lead to undesirable "race conditions," where the program's behavior randomly* depends on which of the processes happens to get to a particular input first.) If there are leftover things to do which don't involve direct communication with the user, you can create extra processes running without windows.

In deciding how many i/o buffers and processes you need, keep in mind that some tasks will be handled by the mouse process, for free. Think back to problem set #2, problem 4, when you defined a `:mouse-click` method which drew a square where the mouse was clicked. That window didn't have a process in it, and you made no use of its i/o buffer. It was the mouse process that watched for user input via the mouse and called your method when appropriate. Similarly, in this graph example, we'll define a `:mouse-click` method to create new nodes at the mouse position (see below), and this will happen independently of our own processes and i/o buffers. The mouse process does it all. There are other tasks, however, for which the mouse process only does some of the work. It turns out that handling clicks on mouse-sensitive items fall into this category. The details are covered later, but what's relevant here is that when a mouse-sensitive item is clicked on, the mouse process simply stuffs something into the i/o buffer of the window under the mouse, and does no more. Somebody else has to notice what's in the i/o buffer and do something about it. That's why the window displaying the graph needs a process.

* See hacker's definition at end of chapter.

5.3 The Windows and the Mouse

The **graph-frame** defflavor

This defflavor uses the `:default-init-plist` option, which lists alternating keywords and values. The effect is as though the keywords and values were present in every make-instance for this flavor (unless explicitly overridden). The `:selected-pane` init keyword means that whenever the graph frame receives a `:select` message, it will pass it on to the the lisp pane. The `:panes` keyword says that this frame has two panes, named "graph" and "lisp," and that they are instances of the flavors **graph-pane** and **tv:lisp-listener-pane**, respectively. The `:configurations` keyword specifies the layout of the frame. Configuration specs are often very messy, and you should try to read up about them. This simple spec states that the graph and lisp panes are stacked vertically, with the graph pane occupying the top 60% of the frame's area, and the lisp pane occupying the rest.

Selection

The next two methods, `:alias-for-selected-windows` and `:selectable-windows`, are ones which applications programmers don't usually mess with. Although what these particular methods do is easy to see, the manner in which they affect the behavior of the Select key and system menus is extremely obscure. The only reason they're here is because I thought it would be nice if we could use the lisp pane to make our frame accessible via Select L. Take a look at the comments in the code, and don't worry about it too much if you don't completely understand. (Incidentally, the bug in the speech editor which I mentioned in the introduction to Part II of the second problem set had to do with these messages.)

The **graph-pane** defflavor

This flavor is based on **graph-window** (defined below), with **tv:pane-no-mouse-select-mixin** added. **tv:pane-no-mouse-select-mixin** itself is just a combination of the flavors **tv:pane-mixin** and **tv:dont-select-with-mouse-mixin**. The former provides the functionality a window needs to be a part of a frame. The latter fixes it so that a pane won't show up in various system menus. Otherwise those menus would contain separate entries for every pane in every frame, which is most inelegant. All you really want is one entry for each frame. (As to exactly what that one entry should be, see the previous paragraph.)

The **graph-window** defflavor

This is the definition of the window in which the graphs are to be displayed. It includes **tv:process-mixin** so that there will be a process running in the window, and **tv:basic-mouse-sensitive-items**, so the window can display items which will be mouse sensitive. We use `:default-init-plist` again here. The `:process` keyword gives the name of a function which will be called on one argument (the window) the first time the window is exposed. Such functions are typically written as loops which never return. The `:item-type-alist` is an instance variable controlling the behavior of the mouse-sensitive items. The contents of this list are described below. The `:blinker-p` keyword specifies that this window is to have no blinkers, and the `:font-map` keyword is a list of fonts this window is to start out knowing about. In this case, the window's sole font will be Helvetica, 12 point italic. (The default font we're all so familiar with is called "cptfont," so the font object may be found as the value of the symbol `fonts:cptfont`.)

The `:init` method for **graph-window**

This is where the variable ***graph-window*** gets set. Note that this setup assumes there is only one active graph-window around at a time. Whenever one is created the old binding of ***graph-window*** is lost, and anybody trying to use the old window is likely to get confused. (If you wished to have several graph-windows around simultaneously, you'd have to think of something more clever than a single global variable to keep track of them. Similarly, the global variable ***all-the-nodes*** would have to go if you wanted different graphs in the different windows.) Note also that this is an after daemon, to avoid clobbering the important `:init` method defined for **tv:sheet**.

The main-loop for **graph-window**'s process

This loop simply reads blips from the window's i/o buffer and dispatches. The only blips it's expecting are ones with a first element of `:typeout-execute`. These blips are generated in the mouse process when someone clicks on a mouse-sensitive item. The remainder of the blip will be a function name, dependent on the type of item and type of click, and the item itself. This loop simply calls the function with two arguments, the item and the graph-window. As we'll soon see, the items will be nodes, and the functions will be ones like **delete-node**, **rename-node**, etc., all defined below.

The `:refresh` method

This generates the picture. It sends each node and each arc the `:draw-self` message. (Note the use of **map-over-arcs**.) This is an after daemon so we don't override the `:refresh` method of **tv:sheet**.

The `:who-line-documentation-string` method

The response a window gives to this message is the string which is displayed in the mouse documentation line at the bottom of the screen whenever the mouse is over the window. `:override` is a kind of method combination we haven't discussed before. It is similar to `:or` combination in that if it returns nil other methods get a chance to run and if it returns anything else they don't. It's different from `:or` combination in that some of the remaining methods may be before or after daemons, which isn't allowed with `:or` combination. In this case, the intent is that if the mouse is not over a mouse-sensitive item, the string supplied in this method should be displayed. But if it is over a mouse-sensitive item, some other method (in particular, the one on flavor **tv:basic-mouse-sensitive-items**) should be allowed to specify the documentation string.

The `:mouse-click` method

If the mouse is not over a mouse-sensitive item, and if the click was a single one on the left button, make a node at the mouse position and display it. Otherwise let the other `:mouse-click` methods have a chance. (Recall that `:mouse-click` methods have `:or` type combination.) The new node will have no label.

The guts of the mouse-sensitive items

For each type of mouse-sensitive item a window knows about, there is a set of possible operations. (Graph-window currently knows about only one type of item, the `:node` type.) The item-type-alist (recall that this is an instance variable of **tv:basic-mouse-sensitive-items**) tells what the possible operations are for each type, and also indicates that one of them is the default operation. The `:mouse-click` method for **tv:basic-mouse-sensitive-items** is set up so that if you click left over an item, a `:typeout-execute` blip is forced into the window's i/o buffer, listing the default operation and the item itself. If you click right, it pops up a menu with all the operations for that type of item, and when you choose one a `:typeout-execute` blip is sent containing the chosen operation and the item. It's up to the process running in the window to read these blips from the i/o buffer and do something with them. As we've just seen, the process in the graph window calls the operation as a function, with arguments of the item and the window.

The internal structure of the alist is a bit of a mess, and it usually is not built by hand. Instead, it is generally constructed with the macro **tv:add-typeout-item-type**, which is called once for each operation defined for an item. The first argument is the alist to be modified, the second is the type of item, the third is a string to name the operation (this appears in the menu you get from clicking right while over an item) and the fourth is the symbol which actually ends up in the blip if this operation is chosen. The fifth and sixth arguments are optional. If the fifth arg is `t`, the operation becomes the default operation for this type of item. If the sixth arg is present, it is a documentation string to appear in the who line while the mouse is over this option in the click-right menu.

You can see that for the graph window the default operation (which you get if you click left over a node) is the function **mouse-new-arc**. If you click right you get a menu listing four other operations in addition to "New Arc," for deleting an arc, moving a node, renaming a node (new label), or deleting a node.

The function definitions for the operations

All that remains are the definitions of the five functions which implement these operations. The first two (**mouse-new-arc** and **mouse-delete-arc**) use a function from the Edit Screen menu to choose an arbitrary point in the window, then see if there's a node under that point, and if so, act accordingly. The **delete-node** function is very simple. The **rename-node** function uses a built-in function named **tv:get-line-from-keyboard** which pops up a small window and prompts the user to type in a line of text. The function for moving a node uses the same piece of Edit Screen as the earlier two.

`:item` and `:primitive-item`

The only thing I haven't explained is how the window knows that some portion of its display is supposed to be a mouse-sensitive item, and of what type. The `:item` and `:primitive-item` messages take care of that. If you look back at the `:draw-self` method for **node**, you'll see the `:primitive-item` message we glossed over before. What this one does is tell the window that there is an item of type `:node`, with left, top, right and bottom edges as given. The window has an instance variable which is a list of all the current mouse-sensitive items. The effect of the `:primitive-item` method is just to push the given item on the list. The `:item` method (which isn't used by graph-window) is an alternative to `:primitive-item` for mouse-sensitive items which are text. The method will take care of printing the item and figuring out its edges. With graphical objects like our nodes it can't do that, so we display them ourselves, calculate the edges, and send the `:primitive-item` message. To see whether there is anything on the list at a given position, send the `:mouse-sensitive-item` message with args of the x and y coords. If there's an item on the list matching those coords, some information about it will be returned, including the item itself, its type, and its edges. This method is primarily for internal use by the `:mouse-click` and `:mouse-moves` methods of **tv:basic-mouse-sensitive-items**, so the window knows to highlight the appropriate region when the mouse is over an item, and what to do if there's a mouse click.

5.4 Fun and Games

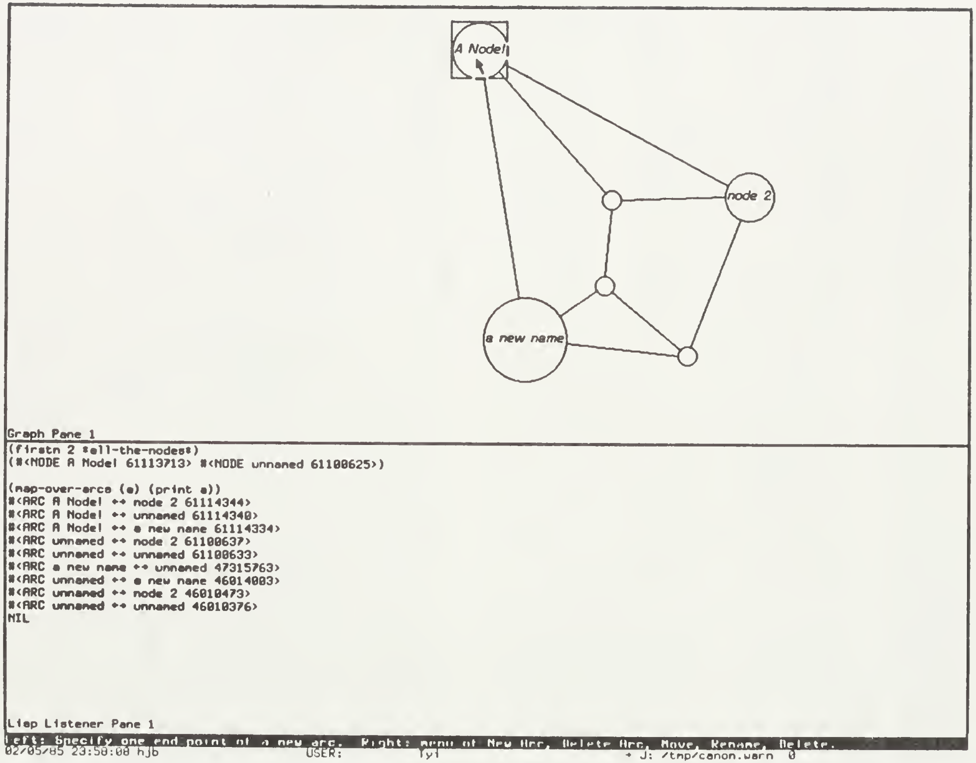
From *The Hacker's Dictionary*, Guy L. Steele, Jr., *et al*:

RANDOM

1. *adjective.* Unpredictable (closest to mathematical definition); weird. "The SYSTEM's been behaving pretty randomly."
2. Assorted; various; undistinguished; uninteresting. "Who was at the conference?" "Just a bunch of random business types."
3. Frivolous; unproductive; undirected. "He's just a random LOSER."
4. Incoherent or inelegant; not well organized. "The program has a random set of MISFEATURES." "That's a random name for that function." "Well, all the names were chosen pretty randomly."
5. Gratuitously wrong; poorly done and for no good apparent reason. "This subroutine randomly uses six registers where two would have sufficed."
6. In no particular order, though deterministic. "The I/O channels are in a pool, and when a file is opened one is chosen randomly."
7. *noun.* A random hacker. This is used particularly of high school students who soak up computer time and generally get in the way. The term "high school random" is frequently heard.
8. One who lives at Random Hall at MIT.

J. RANDOM is often prefixed to a noun to make a "name" out of it (by analogy to common names such as "J. Fred Muggs"). It means roughly "some particular" or "any specific one." The most common uses are "J. Random Loser" and "J. Random Nerd." Example: "Should J. Random Loser be allowed to delete system files without warning?"

5.5 The Program



```
(firstn 2 *all-the-nodes*)
(#<NODE A Node! 61113713> #<NODE unnamed 61100625>)

(map-over-arcs (a) (print a))
#<ARC A Node! ↔ node 2 61114344>
#<ARC A Node! ↔ unnamed 61114340>
#<ARC A Node! ↔ a new name 61114334>
#<ARC unnamed ↔ node 2 61100637>
#<ARC unnamed ↔ unnamed 61100633>
#<ARC a new name ↔ unnamed 47315763>
#<ARC unnamed ↔ a new name 46014003>
#<ARC unnamed ↔ node 2 46010473>
#<ARC unnamed ↔ unnamed 46010376>
NIL
```

```
Left: Specify one end point of a new arc.  Right: menu of New Arc, Delete Arc, Move, Rename, Delete.
02/05/85 23:58:08 hjb          USER:          Ty1          + J: /tmp/canon.warn  0
```

```
;;; -*- Syntax: Zetalisp; Mode: LISP; Package: USER; Base: 10 -*-


;;; first pass at creation and display of undirected graphs

(defvar *all-the-nodes* nil)                    ; list of (active) instances of flavor "node"

(defvar *graph-window* nil)                     ; the window in which the graph is displayed

(defvar *minimum-node-radius* 10)               ; the size (in pixels) of nodes with no label
(defvar *node-margin-size* 3)                   ; pixels between label and perimeter of node

(defflavor node
        ((arcs nil)                             ; list of instances of flavor "arc"
         xpos                                   ; coords of center of node, in pixels
         ypos
         (radius *minimum-node-radius*)         ; in pixels (dependent on length of label)
         (label nil))                           ; a string, to be displayed in the node
        ()
  :settable-instance-variables
  (:required-init-keywords :xpos :ypos))

(defun-method reset-radius node ()
  (setq radius
        (if label
            (max *minimum-node-radius*
                 (+ *node-margin-size*
                    (// (send *graph-window* :string-length label) 2)))
            *minimum-node-radius*)))
```

```
(defmethod (node :init) (&rest ignore)
  ;; set the radius according to the length of the label (in the current font)
  (cond ((not label))
        ((not *graph-window*)
         (tv:notify nil
                    "Warning: node ~S created while there is no graph window; radius not set"
                    self))
        (t (reset-radius)))
  ;; add to the list of all nodes
  (push self *all-the-nodes*))

(defmethod (node :after :set-label) (ignore)
  (and *graph-window* (reset-radius)))

(defmethod (node :print-self) (stream ignore slashify-p)
  (let ((name (or label "unnamed")))
    (if slashify-p
        (SI:PRINTING-RANDOM-OBJECT (SELF STREAM :TYPEP)
          (princ name stream))
        (princ name stream))))

(defmethod (node :draw-self) (window)
  (send window :draw-circle xpos ypos radius)
  (and label
       (send window :display-centered-string label
             (- xpos radius) (+ xpos radius) (- ypos (// (send window :line-height) 2))))
  (send window :send-if-handles
        :primitive-item :node self (- xpos radius) (- ypos radius)
                                   (+ xpos radius) (+ ypos radius)))

(defmethod (node :add-arc) (arc)
  (push arc arcs))

(defmethod (node :remove-arc) (arc)
  (setq arcs (delq arc arcs)))
```

```
(defmethod (node :delete) nil
  ;; (borrowed from tv:sheet's :kill method)
  ;; Do it this way to prevent CDR'ing down list structure being modified
  (LOOP UNTIL (NULL arcs)
        DO (SEND (CAR arcs) :delete))
  (setq *all-the-nodes* (delq self *all-the-nodes*)))


(defflavor arc
        ((mark nil)     ; arbitrary object for misc tagging purposes
         node1          ; the two "node" objects this arc connects
         node2)
        nil
  (:settable-instance-variables mark)
  (:initable-instance-variables node1 node2)
  (:gettable-instance-variables node1 node2))

(defmethod (arc :init) (&rest ignore)
  (send node1 :add-arc self)
  (send node2 :add-arc self))

(defmethod (arc :print-self) (stream &rest ignore)
  (SI:PRINTING-RANDOM-OBJECT (SELF STREAM :TYPEP)
    (format stream "~A ↔ ~A" node1 node2)))

(defmethod (arc :draw-self) (window)
  (multiple-value-bind (x1 y1 x2 y2)
      (find-edges-of-nodes (send node1 :radius) (send node1 :xpos) (send node1 :ypos)
                           (send node2 :radius) (send node2 :xpos) (send node2 :ypos))
    (send window :draw-line x1 y1 x2 y2)))

(defmethod (arc :delete) nil
  (send node1 :remove-arc self)
  (send node2 :remove-arc self))
```

```
(defmacro map-over-arcs ((arc-var) &body body)
  (let ((mark-var (gensym))
        (node-var (gensym)))
    `(loop with ,mark-var = (gensym)
           for ,node-var in *all-the-nodes*
           do (loop for ,arc-var in (send ,node-var :arcs)
                    unless (eq (send ,arc-var :mark) ,mark-var)
                      do ,@body
                         (send ,arc-var :set-mark ,mark-var)))))

(defun find-edges-of-nodes (r1 xpos1 ypos1 r2 xpos2 ypos2)
  (let* ((dx (- xpos2 xpos1))
         (dy (- ypos2 ypos1))
         (length (isqrt (+ (* dx dx) (* dy dy)))))
    (values (+ xpos1 (// (* dx r1) length))
            (+ ypos1 (// (* dy r1) length))
            (- xpos2 (// (* dx r2) length))
            (- ypos2 (// (* dy r2) length)))))



(defflavor graph-frame ()
           (tv:bordered-constraint-frame)
  (:default-init-plist :selected-pane 'lisp
                       :panes '((graph graph-pane)
                                (lisp tv:lisp-listener-pane))
                       :configurations '((main (:layout (main :column graph lisp))
                                               (:sizes (main (graph 0.6)
                                                             :then (lisp :even)))))))

;;; This makes our frame accessible via Select-L, because the Select key now thinks
;;; it's a lisp-listener instead of a graph-frame.  (The method we are overriding is
;;; on basic-frame and would return the graph-frame itself.)
(defmethod (graph-frame :alias-for-selected-windows) ()
  tv:selected-pane)
```

```
;;; Don't ask.  The sneaky thing we just did to make the Select key see the lisp pane at top
;;; level confuses the part of the window system which does things like make the list of
;;; windows for the select option of the system menu.  It would have the lisp pane listed
;;; twice (once for itself and once for the graph-frame).  This fixes it.  (The method we
;;; are overriding is on essential-window.)
(defmethod (graph-frame :selectable-windows) ()
  `((,(send tv:selected-pane :name-for-selection) ,tv:selected-pane)))

(defflavor graph-pane ()
           (tv:pane-no-mouse-select-mixin graph-window))

(defflavor graph-window ()
           (tv:process-mixin
            tv:basic-mouse-sensitive-items
            tv:window)
  (:default-init-plist :process '(graph-window-top-level-function)
                       :item-type-alist *graph-item-type-alist*
                       :blinker-p nil
                       :font-map '(fonts:hl12i)))

(defmethod (graph-window :after :init) (&rest ignore)
  (setq *graph-window* self))

(defun graph-window-top-level-function (window)
  (send window :main-loop))

(defmethod (graph-window :main-loop) nil
  (loop for (keywd op item) = (send self :any-tyi)
        when (eq keywd :typeout-execute)
          do (funcall op item self)))
```

```
(defmethod (graph-window :after :refresh) (&rest ignore)
  (loop for node in *all-the-nodes*
        do (send node :draw-self self))
  (map-over-arcs (arc)
    (send arc :draw-self self)))

(defmethod (graph-window :override :who-line-documentation-string) nil
  (and (not (send self :mouse-sensitive-item tv:mouse-x tv:mouse-y))
       "L:  Make a new node,      R2:  System menu"))

(defmethod (graph-window :mouse-click) (buttons x y)
  (and (not (send SELF ':MOUSE-SENSITIVE-ITEM X Y))     ; do nothing when over m-s items
       (= buttons #\mouse-1-1)
       (progn (send (make-instance 'node :xpos (- x tv:left-margin-size)
                                         :ypos (- y tv:top-margin-size))
                    :draw-self self)
              t)))

(defvar *graph-item-type-alist* nil)

(tv:add-typeout-item-type *graph-item-type-alist* :node "Delete" delete-node nil
                          "Splice this node out of the graph.")

(tv:add-typeout-item-type *graph-item-type-alist* :node "Rename" rename-node nil
                          "Provide a new label for this node.")

(tv:add-typeout-item-type *graph-item-type-alist* :node "Move" mouse-move-node nil
                          "Specify a new position for this node.")

(tv:add-typeout-item-type *graph-item-type-alist* :node "Delete Arc" mouse-delete-arc nil
                          "Specify one end point of an arc to be deleted.")

(tv:add-typeout-item-type *graph-item-type-alist* :node "New Arc" mouse-new-arc t
                          "Specify one end point of a new arc.")
```

```
(defun mouse-new-arc (node window)
  (multiple-value-bind (x y)
      (tv:mouse-set-sheet-then-call window
        #'tv:screen-editor-find-point nil "Pick Point"
                                          "Choose the other node for the new arc.")
    (let ((item (send window :mouse-sensitive-item x y)))
      (if (not item) (beep)
          (send (make-instance 'arc :node1 node :node2 (second item))
                :draw-self window)))))

(defun mouse-delete-arc (node window)
  (multiple-value-bind (x y)
      (tv:mouse-set-sheet-then-call window
        #'tv:screen-editor-find-point nil "Pick Point"
                                          "Show the other node for the arc to be deleted.")
    (let ((item (send window :mouse-sensitive-item x y)))
      (if (not item) (beep)
          (let ((arc (loop for a in (send (second item) :arcs)
                           when (or (eq node (send a :node1))
                                    (eq node (send a :node2)))
                           return a)))
            (if (not arc) (beep)
                (send arc :delete)
                (send window :refresh)))))))

(defun delete-node (node window)
  (send node :delete)
  (send window :refresh))

(defun rename-node (node window)
  (send node :set-label (tv:get-line-from-keyboard "Enter a label for the node"
                                                   tv:mouse-sheet #'readline-or-nil))
  (send window :refresh))
```

```
(defun mouse-move-node (node window)
  (multiple-value-bind (x y)
      (tv:mouse-set-sheet-then-call window
        #'tv:screen-editor-find-point nil "Pick Point" "Choose a new position for the node.")
    (send node :set-xpos (- x (send window :left-margin-size)))
    (send node :set-ypos (- y (send window :top-margin-size))))
  (send window :refresh))
```

5.6 Problem Set #5

Questions

1. Assure yourself that you understand the graph code.

2. Write a function that finds any nodes with no connections to other nodes and removes them from the graph. Write another which clears the graph-window then uses **map-over-arcs** to display only those nodes which are "node1" for some arc.

3. Extend the program to be able to deal with a larger area than will fit on the window at one time. You'll need some mechanism for moving the graph window around within the entire space.

4. A. Write a `:highlight` method for nodes, which puts an already displayed node in reverse-video. Now modify the functions **mouse-new-arc** and **mouse-delete-arc** to highlight the first node that was moused while the program is waiting for you to choose the second, and un-highlight it when you've chosen. Fix up **mouse-move-node** similarly.

   B. Write a new mouse-sensitive operation which has you pick two nodes and then highlights all the nodes in the shortest path between the two chosen ones (where "shortest" simply means containing the fewest nodes — don't worry about ties).

5. (*Hard.*) Make the arcs mouse-sensitive too, but modify the mechanism so that the area considered part of the item is not an entire rectangle, because that would include too much area. Allow specification of parallelograms, or something like that, so you can tighten the mouse-sensitive area to be just around the arc.

6. There are a number of problems with the code as it stands. Here's your chance to improve on the teacher's work.

   A. Any operation which requires that some part of the display be erased currently causes a complete redisplay. This is really unnecessary, and quite unattractive, especially if there's a lot of stuff on the screen. Fix the deletion commands to do minimal redisplay.

   B. Currently, if two nodes are connected by an arc which cuts across another node, the line for the arc just runs right through the node. Fix the `:draw-self` method for arcs to be smart enough to go around obstacles.

7. Write a new program, borrowing whatever portions of this one are

appropriate, for displaying trees. One important difference should be that instead of having the user specify the location of each node, your program will determine their locations. That is, the root will appear at the top with its inferiors spread out in some tasteful fashion below it.

Hints

1. Play.

2. The first function should loop through all the nodes, sending the `:delete` message to any which have no arcs, and then refresh the graph window. The second should clear the graph window, then use **map-over-arcs** to loop through all the arcs, sending "node1" of each arc the `:draw-self` message.

3. Give the **graph-window** flavor two new instance variables to indicate the x and y position of the window relative to the graph area. Now the functions concerned with node positions need to convert between a node's apparent position and its real position by adjusting for the window position. Four of graph-window's methods (`:draw-circle`, `:draw-line`, `:display-centered-string`, `:primitive-item`) will have to convert from real node positions to apparent positions, while the `:mouse-click` method and the function **mouse-move-node** will have to convert from apparent to real. The former four are all inherited from some other flavor, so for them the easiest thing is probably to define whoppers which adjust the arguments. For the latter two, on the other hand, we did the relevant definitions ourselves, so we can change them.

   The whoppers converting from real positions to apparent positions should be prepared for the apparent position being off the screen entirely. The `:draw-circle` and `:draw-line` methods "clip" (they'll even do the right thing if the circle or line should be partially visible), so you can just pass along any bogus arguments without worry. The `:display-centered-string` method, however, wraps around when given off-screen coordinates, so in this case you should check the apparent position and possibly skip the whopper continuation.

   (You'll have to make a new instance of the graph-frame, because the new **graph-window** defflavor, with two extra instance variables, is incompatible with the existing graph-window.)

4. A. The `:highlight` method should draw a filled-in circle in xor mode. The basic idea for the mouse-[...] functions is to use the `:highlight` method once to highlight a node and again to turn it off, but there are two kinds of complications to watch out for that could leave you with a node turned on: the `:refresh` method clears the screen, thus removing any highlighting that may be present; the **new-arc** and **delete-arc** functions have several failure modes (the parts with the beeps) that skip portions of the code.

   B. Add a new type of typeout item to call the function **mouse-find-path**. The control structure of **mouse-find-path** should look roughly like that of **mouse-delete-arc**, and you'll probably want an auxiliary recursive function, say **find-path-to**. A breadth-first search will be the easiest way to find the shortest path (because the first path you find will be the shortest); you can use the `mark` instance variable of arcs to prevent looping in the search.

5. The key issue here, and the reason I call this a hard problem, is compatibility. We want to use the bulk of the existing mouse-sensitive-item code to save us the trouble of duplicating its functionality, and we want to modify it to provide the new features. But at the same time, other programs will be using the ms-item code and counting on it to behave the way it used to.

   My approach is to first replace the **hollow-rectangular-blinker** that is provided by the **basic-mouse-sensitive-items** mixin with a **polygonal-blinker**, a new flavor of blinker which can draw itself as any closed polygon (imitating hollow-rectangular-blinkers as a special case). Then I define a **line-item** object, and arrange for the polygonal-blinker to draw itself as a squashed hexagon around ms items which are of type line-item (and as a rectangle around all other ms items). A number of basic-mouse-sensitive-items methods have to be modified, but most of the changes are of one very simple kind: references to individual ms items are replaced with calls to a macro which just returns the item if it's the normal type, and extracts the appropriate information out of it if it's a line-item. The only substantive changes are to the `:mouse-moves` method, because there's a new way for specifying the shape of the blinker, and the **typeout-mouse-item** defun-method, because there are new rules for determining whether the mouse is over a particular item.

   Now all I have to do is change the `:draw-self` method for arcs to provide a line-item as an argument to the `:primitive-item` message, and define the "Delete" operation on mouse-sensitive arcs.

   (You'll have to make a new graph-frame to put all the changes into effect.)

6. A. The deletion commands should surgically erase exactly the nodes and/or arcs being deleted, and leave the rest alone. You can erase the nodes by drawing a filled-in circle in andca mode, and the arcs by drawing over them in xor or andca mode.

   B. Open problem. I haven't thought of a good way to do this.

7. Done as "The Tree Example," chapter 7.

Problem Set #5 -- Solutions

2)
```
(defun remove-loners ()
  (loop for node in *all-the-nodes*
        when (null (send node :arcs))
          do (send node :delete))
  (send *graph-window* :refresh))

(defun display-node1s ()
  (send *graph-window* :clear-window)
  (map-over-arcs (arc)
    (send (send arc :node1) :draw-self *graph-window*)))
```

3) The following provides all the needed functionality, but requires that you send the window :set-graph-x-pos and :set-graph-y-pos messages to move it around over the graph. It would be more convenient to scroll with the mouse. You might want to look into the mixins tv:basic-scroll-bar and tv:flashy-scrolling-mixin.

```
(defflavor graph-window
        ((graph-x-pos 0)                    ; these two added and made settable
         (graph-y-pos 0))
        (tv:process-mixin
         tv:basic-mouse-sensitive-items
         tv:window)
  :settable-instance-variables
  (:default-init-plist :process '(graph-window-top-level-function)
                       :item-type-alist *graph-item-type-alist*
                       :blinker-p nil
                       :font-map `(,fonts:hl12i)))

 (defwhopper (graph-window :draw-circle) (x y radius)
   (continue-whopper (- x graph-x-pos) (- y graph-y-pos) radius))
```

```
(defwhopper (graph-window :display-centered-string) (string left right ypos)
  (decf left graph-x-pos) (decf right graph-x-pos) (decf ypos graph-y-pos)
  (and (plusp left)
       (< right (send self :inside-width))
       (< 0 ypos (- (send self :inside-height) tv:line-height))
       (continue-whopper string left right ypos)))

(defwhopper (graph-window :primitive-item) (type item left top right bottom)
  (continue-whopper type item (- left graph-x-pos) (- top graph-y-pos)
                              (- right graph-x-pos) (- bottom graph-y-pos)))

(defwhopper (graph-window :draw-line) (x1 y1 x2 y2)
  (continue-whopper (- x1 graph-x-pos) (- y1 graph-y-pos)
                    (- x2 graph-x-pos) (- y2 graph-y-pos)))

(defmethod (graph-window :mouse-click) (buttons x y)
  (and (not (send SELF ':MOUSE-SENSITIVE-ITEM X Y))
       (= buttons #\mouse-1-1)
       (progn (send (make-instance 'node :xpos (+ (- x tv:left-margin-size) graph-x-pos)
                                         :ypos (+ (- y tv:top-margin-size) graph-y-pos))
                    :draw-self self)
              t)))

(defun mouse-move-node (node window)
  (multiple-value-bind (x y)
      (tv:mouse-set-sheet-then-call window
        #'tv:screen-editor-find-point nil "Pick Point"
                                          "Choose a new position for the node.")
    (send node :set-xpos (+ (- x (send window :left-margin-size))
                            (send window :graph-x-pos)))
    (send node :set-ypos (+ (- y (send window :top-margin-size))
                            (send window :graph-y-pos))))
  (send window :refresh))
```

4) A. Highlighting

```
(defmethod (node :highlight) (window)
  (send window :draw-filled-in-circle xpos ypos radius tv:alu-xor))

(defun mouse-new-arc (node window)
  (send node :highlight window)              ; changed
  (multiple-value-bind (x y)
      (tv:mouse-set-sheet-then-call window
        #'tv:screen-editor-find-point nil "Pick Point"
        "Choose the other node for the new arc.")
    (let ((item (send window :mouse-sensitive-item x y)))
      (if (not item) (beep)
          (send (make-instance 'arc :node1 node :node2 (second item))
                :draw-self window))))
  (send node :highlight window))             ; changed

(defun mouse-delete-arc (node window)
  (send node :highlight window)              ; changed
  (multiple-value-bind (x y)
      (tv:mouse-set-sheet-then-call window
        #'tv:screen-editor-find-point nil "Pick Point"
        "Show the other node for the arc to be deleted.")
    (let ((item (send window :mouse-sensitive-item x y)))
      (if (not item) (progn (beep) (send node :highlight window))   ; changed
          (let ((arc (loop for a in (send (second item) :arcs)
                           when (or (eq node (send a :node1))
                                    (eq node (send a :node2)))
                            return a)))
            (if (not arc) (progn (beep) (send node :highlight window))      ; changed
                (send arc :delete)
                (send window :refresh))))))
  ;; can't just put a :highlight here because in the case where we successfully
  ;; deleted an arc, we did a :refresh, which itself clears the highlighting
  )
```

```
(defun mouse-move-node (node window)
  (send node :highlight window)              ; changed
  (multiple-value-bind (x y)
      (tv:mouse-set-sheet-then-call window
        #'tv:screen-editor-find-point nil "Pick Point"
                                          "Choose a new position for the node.")
    (send node :set-xpos (- x (send window :left-margin-size)))
    (send node :set-ypos (- y (send window :top-margin-size))))
  (send window :refresh))
```

B. Path-finding

```
(tv:add-typeout-item-type *graph-item-type-alist* :node "Find Path" mouse-find-path nil
                          "Find the shortest path between this node and another one.")

(defun mouse-find-path (node window)
  (send node :highlight window)
  (multiple-value-bind (x y)
      (tv:mouse-set-sheet-then-call window
        #'tv:screen-editor-find-point nil "Pick Point"
        "Choose the node to which you wish to see the shortest path.")
    (let ((item (send window :mouse-sensitive-item x y)))
      (if (not item) (progn (beep) (send node :highlight window))
          (let ((path (find-path-to (second item) `((,node)) (gensym))))
            (if path (loop for n in (cdr path) do (send n :highlight window))
                (progn (beep) (send node :highlight window))))))))
```

```
(defun find-path-to (target partial-paths mark)
  (and partial-paths
       (block path-finding
         (find-path-to
           target
           (loop for path in partial-paths
                 for end-node = (car path)
                 nconcing (loop for arc in (send end-node :arcs)
                                for new-node = (send arc :other-node end-node)
                                for new-path = (cons new-node path)
                                when (eq new-node target)
                                  do (return-from path-finding (nreverse new-path))
                                unless (eq (send arc :mark) mark)
                                  collect new-path and do (send arc :set-mark mark)))
           mark))))

(defmethod (arc :other-node) (node)
  (if (eq node node1) node2
      (if (eq node node2) node1
          nil)))
```

Alternative path-finding: rather than have the queue of partial paths implicitly maintained in the stack by the recursive control structure, we can have an explicit queue. find-path-to would then iteratively ask the queue for the next item, and the queue would decide which partial path should be pursued next. One advantage of this approach is that different kinds of searches (depth-first, breadth-first, best-first, etc.) can be implemented just by tweaking the :add-item and :get-next queue methods: for depth-first you push new items on, and pop them off, the front of the queue list; for breadth-first you splice new items onto the back of the list, and pop them off the front; for best-first you sort the list before returning the next item. For all cases, the code which makes use of the queue (find-path-to) is identical.

```
(defflavor queue
        ((in-ptr nil)
         (out-ptr nil))
        ())

(defmethod (queue :add-item) (item)
  (if (null in-ptr)
      (setq in-ptr (setq out-ptr (ncons item)))
      (rplacd out-ptr (setq out-ptr (ncons item))))
  item)

(defmethod (queue :get-next) ()
  (pop in-ptr))

replace the following line in mouse-find-path
  (let ((path (find-path-to (second item) `((,node)) (gensym))))
with
  (let ((queue (make-instance 'queue)))
    (send queue :add-item `(,node))
    (let ((path (find-path-to (second item) queue (gensym))))

(defun find-path-to (target queue mark)
  (loop named path-found
        for path = (send queue :get-next)
        while path
        do (loop with node = (car path)
                 for arc in (send node :arcs)
                 for new-node = (send arc :other-node node)
                 for new-path = (cons new-node path)
                 when (eq new-node target)
                   do (return-from path-found (nreverse new-path))
                 unless (eq (send arc :mark) mark)
                   do (send arc :set-mark mark)
                      (send queue :add-item new-path))))
```

```
5)  tv:(compiler-let ((package (pkg-find-package "tv")))

    (DEFFLAVOR polygonal-BLINKER
            (points)                                ; list of x,y pairs (outside coords)
            (BLINKER)
      :settable-instance-variables)

    (defmethod (polygonal-blinker :size) nil
      (values 0 0))                                 ; required method - I'm not sure who calls it

    ;;; modeled after blinker's :set-cursorpos method, to handle "opening" of
    ;;; blinker properly
    (DEFMETHOD (polygonal-BLINKER :SET-points) (list-of-pairs &AUX (OLD-PHASE PHASE))
      (setq list-of-pairs (loop for (x . y) in list-of-pairs
                                collect (cons (fix x) (fix y))))
      (WITH-BLINKER-READY T
        (COND ((NULL VISIBILITY)                    ; Don't open if visibility NIL
               (SETQ points list-of-pairs FOLLOW-P NIL))
              ((not (equal points list-of-pairs))     ; Only blink if actually moving blinker
               (OPEN-BLINKER SELF)
               (SETQ points list-of-pairs FOLLOW-P NIL)
               ;; If this is the mouse blinker, and it is not being tracked by microcode,
               ;; then it is important to turn it back on immediately.
               (AND (NEQ VISIBILITY ':BLINK)
                    OLD-PHASE
                    (BLINK SELF))))))

    (defmethod (polygonal-blinker :blink) nil
      (loop with (first-x . first-y) = (car points)
            for prev-x = first-x then x
            for prev-y = first-y then y
            for (x . y) in (cdr points)
            do (sheet-draw-line prev-x prev-y x y alu-xor nil sheet)
            finally (sheet-draw-line prev-x prev-y first-x first-y
                                     alu-xor nil sheet)))
```

```
(defstruct (line-item :named :conc-name)
  item
  radius
  orientation)

(defmacro extract-item (item-ref)
  (once-only (item-ref)
    `(if (typep ,item-ref 'line-item) (line-item-item ,item-ref) ,item-ref)))

;;;Make a blinker for the menu type items and the pop-up menu
(DEFMETHOD (BASIC-MOUSE-SENSITIVE-ITEMS :AFTER :INIT) (IGNORE)
  (SETQ ITEM-BLINKER (MAKE-BLINKER SELF 'polygonal-BLINKER ':VISIBILITY NIL)
        MENU (MAKE-WINDOW 'MOMENTARY-MENU ':SUPERIOR SELF)))

;;;Type out item, either as itself or FORMAT-ARGS.  TYPE is used for indexing
;;;into ITEM-TYPE-ALIST
(DEFMETHOD (BASIC-MOUSE-SENSITIVE-ITEMS :ITEM) (TYPE ITEM &REST FORMAT-ARGS)
  (LET ((X CURSOR-X))
    (IF FORMAT-ARGS (LEXPR-FUNCALL #'FORMAT SELF FORMAT-ARGS)
        (PRINC (extract-item ITEM) SELF))
    (PUSH (LIST TYPE ITEM X CURSOR-Y CURSOR-X (+ CURSOR-Y LINE-HEIGHT))
          ITEM-LIST)
    (MOUSE-WAKEUP)))
```

```
;;;Blink any item the mouse points to
(DEFMETHOD (BASIC-MOUSE-SENSITIVE-ITEMS :MOUSE-MOVES) (X Y &AUX ITEM)
  (MOUSE-SET-BLINKER-CURSORPOS)
  (COND ((AND (SETQ ITEM (SEND SELF ':MOUSE-SENSITIVE-ITEM X Y))
              (ASSQ (TYPEOUT-ITEM-TYPE ITEM) ITEM-TYPE-ALIST))
         (LET ((LEFT (TYPEOUT-ITEM-LEFT ITEM))
               (TOP (TYPEOUT-ITEM-TOP ITEM))
               (RIGHT (TYPEOUT-ITEM-RIGHT ITEM))
               (BOTTOM (TYPEOUT-ITEM-BOTTOM ITEM))
               (item (typeout-item-item item))
               BWIDTH BHEIGHT)
           (SETQ BWIDTH (- RIGHT LEFT)
                 BHEIGHT (- BOTTOM TOP))
           (if (typep item-blinker 'polygonal-blinker)
               (send item-blinker :set-points
                     (if (typep item 'line-item)
                         (let* ((radius (line-item-radius item))
                                (orientation (line-item-orientation item))
                                (length (sqrt (+ (* bwidth bwidth) (* bheight bheight))))
                                (dx (// (* radius length) bheight))
                                (dy (// (* radius length) bwidth)))
                           (when (eq orientation :\)
                             (swapf top bottom)
                             (setq dy (- dy)))
                           `((,left . ,bottom) (,(+ left dx) . ,bottom)
                             (,right . ,(+ top dy)) (,right . ,top)
                             (,(- right dx) . ,top) (,left . ,(- bottom dy))))
                         `((,left . ,top) (,right . ,top)
                           (,right . ,bottom) (,left . ,bottom))))
               ;; else is old type (will happen with some pre-existing windows)
               (BLINKER-SET-CURSORPOS ITEM-BLINKER (- LEFT (SHEET-INSIDE-LEFT))
                                      (- TOP (SHEET-INSIDE-TOP)))
               (BLINKER-SET-SIZE ITEM-BLINKER BWIDTH BHEIGHT))
           (BLINKER-SET-VISIBILITY ITEM-BLINKER T)))
        (T (BLINKER-SET-VISIBILITY ITEM-BLINKER NIL))))
```

```
;;;Mouse-left selects the blinking item, mouse-right pops up a menu near it
(DEFMETHOD (BASIC-MOUSE-SENSITIVE-ITEMS :MOUSE-CLICK) (BUTTON X Y &AUX ITEM)
  (COND ((SETQ ITEM (SEND SELF ':MOUSE-SENSITIVE-ITEM X Y))
         (LET ((ITEM-TYPE (TYPEOUT-ITEM-TYPE ITEM)))
           (COND ((SETQ ITEM-TYPE (ASSQ ITEM-TYPE ITEM-TYPE-ALIST))
                  (SELECTQ BUTTON
                    (#\MOUSE-1-1
                     (SEND SELF ':FORCE-KBD-INPUT
                           (LIST ':TYPEOUT-EXECUTE (CADR ITEM-TYPE)
                                 (extract-item (TYPEOUT-ITEM-ITEM ITEM))))
                     T)
                    (#\MOUSE-3-1
                     (PROCESS-RUN-FUNCTION "Menu Choose" #'TYPEOUT-MENU-CHOOSE
                                           MENU (CDDDR ITEM-TYPE) ITEM SELF)
                     T))))))))
```

```
;;;Return the item the mouse if pointing to
(DEFUN-METHOD TYPEOUT-MOUSE-ITEM BASIC-MOUSE-SENSITIVE-ITEMS (X Y)
  (DO ((ITEMS ITEM-LIST (CDR ITEMS))
       (ITEM) (ITEM-Y) (WRAPPED-AROUND))
      ((NULL ITEMS))
    (IF (SYMBOLP (SETQ ITEM (CAR ITEMS)))
        (SETQ WRAPPED-AROUND T)
        (AND (≤ (SETQ ITEM-Y (TYPEOUT-ITEM-TOP ITEM)) CURSOR-Y) WRAPPED-AROUND
             (RETURN NIL))
        (AND (≥ Y ITEM-Y)
             (< Y (TYPEOUT-ITEM-BOTTOM ITEM))
             (≥ X (TYPEOUT-ITEM-LEFT ITEM))
             (< X (TYPEOUT-ITEM-RIGHT ITEM))
             (or (not (typep (typeout-item-item item) 'line-item))
                 (let* ((line-item (typeout-item-item item))
                        (radius (line-item-radius line-item))
                        (orientation (line-item-orientation line-item)))
                   (≤ (point-to-line
                        x y
                        (typeout-item-left item) (if (eq orientation :\)
                                                     (typeout-item-top item)
                                                     (typeout-item-bottom item))
                        (typeout-item-right item) (if (eq orientation :\)
                                                      (typeout-item-bottom item)
                                                      (typeout-item-top item)))
                        radius)))
             (RETURN ITEM)))))

;;; find the distance between a point (px,py) and a line [(x1,y1)-(x2,y2)]
(defun point-to-line (px py x1 y1 x2 y2)
  (decf px x1) (decf py y1)
  (decf x2 x1) (decf y2 y1)
  (let ((temp (- (* px y2) (* x2 py))))
    (sqrt (// (* temp temp)
              (+ (* x2 x2) (* y2 y2))))))
```

```
;;;Select thing to do with selected item from menu
(DEFUN TYPEOUT-MENU-CHOOSE (MENU ALIST TYPEOUT-ITEM TYPEOUT-WINDOW)
  ;; Unadvertizable kludge
  (SEND MENU ':SET-LABEL
        (AND (INSTANCEP (extract-item (TYPEOUT-ITEM-ITEM TYPEOUT-ITEM)))
             (SEND (extract-item (TYPEOUT-ITEM-ITEM TYPEOUT-ITEM))
                   ':SEND-IF-HANDLES ':STRING-FOR-PRINTING)))
  (SEND MENU ':SET-ITEM-LIST ALIST)
  (MOVE-WINDOW-NEAR-RECTANGLE MENU
                              (TYPEOUT-ITEM-LEFT TYPEOUT-ITEM)
                              (TYPEOUT-ITEM-TOP TYPEOUT-ITEM)
                              (TYPEOUT-ITEM-RIGHT TYPEOUT-ITEM)
                              (TYPEOUT-ITEM-BOTTOM TYPEOUT-ITEM))
  (LET ((CHOICE-RESULT (SEND MENU ':CHOOSE)))
    (AND CHOICE-RESULT
         (SEND TYPEOUT-WINDOW ':FORCE-KBD-INPUT
               (LIST ':TYPEOUT-EXECUTE CHOICE-RESULT
                     (extract-item (TYPEOUT-ITEM-ITEM TYPEOUT-ITEM)))))))

)

(defmethod (arc :draw-self) (window)
  (multiple-value-bind (x1 y1 x2 y2)
      (find-edges-of-nodes (send node1 :radius) (send node1 :xpos) (send node1 :ypos)
                           (send node2 :radius) (send node2 :xpos) (send node2 :ypos))
    (send window :draw-line x1 y1 x2 y2)
    (send window :send-if-handles :primitive-item
          :arc (tv:make-line-item
                 tv:item self
                 tv:radius 10.
                 tv:orientation (if (plusp (* (- x2 x1) (- y2 y1))) :\ ://))
          (min x1 x2) (min y1 y2) (max x1 x2) (max y1 y2))))

(tv:add-typeout-item-type *graph-item-type-alist* :arc "Delete" mouse-delete-this-arc t
                          "Remove this arc from the graph.")
```

```
(defun mouse-delete-this-arc (arc window)
  (send arc :delete)
  (send window :refresh))
```

Chapter 6

STREAMS AND FILES

6.1 Streams

Since the mechanics of interacting with different kinds of peripheral devices vary widely, and are often quite messy, it is desirable to shield programmers from having to know the details of such operations. This shielding is accomplished by routing all input and output operations through *streams*. A stream is a message-receiving object (some use the flavor system and some do not). There are different kinds of streams for the different kinds of peripherals. All streams accept generic commands to perform some operation, and take care themselves of the details of performing that operation on their particular sort of device. This way, knowledge about how to perform I/O operations is segregated into the streams themselves, freeing programs (and programmers) from the need to understand the details of these operations. All a program needs to know is how to deal with streams; the streams know how to deal with everything else.

6.1.1 General Purpose Stream Operations (3.2, Volume 5)

Some streams only handle input; some only handle output; some do both. There is a small set of basic operations that all output streams are required to handle.

Similarly, there is another set that all input streams are required to handle. Additionally, there is a somewhat larger set that all streams are guaranteed to accept, even though they themselves may not handle them. This bit of magic works through the *default handler*. Whenever a stream receives a message it has no handler for, it passes the message on to the default handler. The default handler then tries to use some combination of messages the stream does handle to produce the desired effect. For instance, the **:tyo** operation is required of all output streams. It outputs a single character. The **:string-out** operation, which outputs a string of characters, is in the set that is guaranteed via the default handler. Some streams handle this operation directly; for the ones that don't and pass it on to the default handler, it achieves the same effect (albeit more slowly) by repeatedly sending the stream the **:tyo** message.

Among the messages all streams handle is **:which-operations**. The list returned includes only those messages handled directly by the stream. Messages handled by the default-handler on behalf of the stream will not appear in the list.

Here are some of the more commonly used messages which are accepted by all streams of the appropriate type (input or output), possibly via the default handler.

:tyo *char*

The stream will output the character *char*. For example, if `s` is bound to a stream, then `(send s :tyo #\B)` outputs a "B" on the stream. (Recall that "#\" is a reader macro that expands into the fixnum representation of the character code for the given character.)

:tyi &optional *eof*

The stream will input one character and return it (as a fixnum). On an interactive device this is likely to mean first waiting for input to become available. The optional *eof* argument tells the stream what to do if it gets to the end of the file (however end-of-file is defined for that kind of stream). If the argument is not provided, or is nil, the stream will return nil at the end of file. Otherwise it will signal an error, and print out the argument as the error message. (This is *not* the same as the eof optional argument to **read**, **tyi**, and related functions.)

:untyi *char*

The stream will remember the character *char*, and the next time an input character is requested (presumably via **:tyi**), the stream will return *char*. Some restrictions: you are only allowed to **:untyi** one character before doing a **:tyi**, and you aren't allowed to **:untyi** a different character than the last character you read from the stream. Some streams implement **:untyi** by saving the character, while others back up the pointer into a buffer.

:characters
Returns `t` if the stream is a character stream, `nil` if it is a binary stream.

:direction
Returns one of the keyword symbols `:input`, `:output`, or `:bidirectional`.

:listen
This is a test to see whether there is any input waiting to be read. On an interactive device, **:listen** returns non-nil if there are any input characters immediately available, or nil if there is no immediately available input. On a non-interactive device, the operation always returns non-nil except at end-of-file.

:tyipeek &optional *eof*
Returns the next character that is about to be read (without removing it from the input buffer), or nil if the stream is at end-of-file. The *eof* argument is interpreted as for **:tyi**. **:tyipeek** is defined to have the same effect as a **:tyi** followed by an **:untyi** (if end-of-file was not reached), meaning that you may not read some character, do a **:tyipeek** to look at the next character, and then **:untyi** the original character.

:string-out *string* &optional *start end*
The characters of the string are successively output to the stream. Many streams can perform this operation much more efficiently than the equivalent sequence of **:tyo** operations. If *start* and *end* are not supplied, the whole string is output. Otherwise a substring is output; *start* is the index of the first character to be output (defaulting to 0), and *end* is one greater than the index of the last character to be output (defaulting to the length of the string).

:string-in *eof-option string* &optional *start end*
The stream inputs a number of characters, reading them into *string*. *string* may be any kind of array, not necessarily a string, which can be very handy for reading from binary files. *start* and *end* specify the substring to use, defaulting to the entire string. *eof* specifies what to do if end-of-file is encountered before reading the intended number of characters. If nil, **:string-in** returns normally and sets the fill-pointer of *string* (if it has one) to point just beyond the last character read. If non-nil, the condition **sys:end-of-file** is signaled, with the value of *eof* as the report string.

:clear-input
The stream clears any buffered input, *i.e.*, input sitting in its buffer, waiting to be read.

:clear-output
The stream clears any buffered output.

:force-output
This is for output streams to buffered asynchronous devices, such as the Chaosnet. Any buffered output is sent to the device. **:force-output** returns immediately, without waiting for the output to be completed. For that, use **:finish**. If a stream supports **:force-output**, then usage of **:tyo**, **:string-out**, and the like, may have no visible effect until a **:force-output** is done.

:finish
The stream does a **:force-output** then waits for the output to complete before returning.

:close &optional *mode*
The stream is "closed" and no further operations should be performed on it. If *mode* is `:abort`, and the stream is outputting to a file, and it has not been closed already, the stream's newly-created file will be deleted, as though it had never been opened.

6.1.2 Special Purpose Operations (3.3, Volume 5)

There are a wide variety of operations particular to streams for one or another of the peripherals (files, Chaosnet, windows, etc). Most of these are not handled by the default handler, and so would result in an error if sent to a stream which does not itself handle them. The bulk of the special-purpose operations are documented along with the type of device they're intended to be used with. Here are a few of the more commonly-used of these operations.

:tyi-no-hang &optional *eof*
Just like **:tyi** except that if it would be necessary to wait in order to get the character, returns nil instead.

:read-cursorpos &optional (*units* `:pixel`)
This operation is supported by windows. It returns two values: the current *x* and *y* coordinates of the cursor. The optional argument is a symbol indicating in what units *x* and *y* should be expressed; the symbols `:pixel` and `:character` are understood.

:set-cursorpos *x y* &optional (*units* `:pixel`)

This operation is supported by the same streams that support **:read-cursorpos**. It sets the position of the cursor. *x* and *y* are like the values of **:read-cursorpos** and *units* is the same as the *units* argument to **:read-cursorpos**.

:clear-window (`:clear-screen` in older code)

Erases the screen area on which this stream displays.

:read-pointer

This operation, and the next, are supported by streams to random-access devices, principally files. **:read-pointer** returns the current position within the file, expressed in either 8-bit or 16-bit bytes, depending on the type of the stream.

:set-pointer *new-pointer*

Sets the reading position within the file to *new-pointer*, where the units are as with **:read-pointer**. This operation is for input streams only.

6.1.3 Standard Streams (3.4, Volume 5)

There are about half a dozen special variables whose values are streams widely used by many system (as well as user) functions. Here are some of them.

standard-input

In the normal Lisp top-level loop, input is read from **standard-input** (*i.e.*, whatever stream is the value of **standard-input**). Many input functions, including **tyi** and **read**, take a stream argument which defaults to **standard-input**.

standard-output

Analogous to **standard-input**; in the Lisp top-level loop, output is sent to the stream which is the value of **standard-output**, and many output functions, including **tyo** and **print**, take a stream argument which defaults to **standard-output**.

terminal-io

The value of **terminal-io** is the stream which connects to the user's console. In a process which is running in a window (keep in mind that each process has its own binding stack, and thus can have its own value for a given special variable), the value of **terminal-io** is likely to be that window. For processes without windows, which don't normally communicate directly with the user (like the mouse process), **terminal-io** defaults to a stream which does not expect to ever be used. If it is used, perhaps by an error printout, it turns into a "background" window and requests the user's attention.

error-output
The value of **error-output** is a stream to which error messages should be sent. Normally this is the same as **standard-output**, but **standard-output** might be bound to a file and **error-output** left pointing to the terminal.

standard-input, **standard-output** and **error-output** are initially bound to *synonym streams* which pass all operations on to the stream which is the value of **terminal-io**. That is, they are bound to an uninterned symbol whose function definition is forwarded to the value cell of **terminal-io**. So if **terminal-io** is re-bound (*i.e.*, the contents of its value cell change) the synonym streams see the new value.

User programs generally don't change the value of **terminal-io**. A program which wants, for example, to divert output to a file should do so by temporarily binding **standard-output**; that way error messages sent to **error-output** can still get to the user by going through **terminal-io**, which is usually what is desired.

6.1.4 Making Your Own Streams (3.5, Volume 5)

While most streams are actually instances of some flavor, and handle their messages through the usual message-handling mechanism of calling the appropriate method, all that's really needed for a simple stream is a function which dispatches off its first argument (the operation) and calls the default handler if it doesn't recognize the operation. Here's a simple output stream, which accepts characters and conses them onto a list:

```
(defvar the-list nil)

(defun list-output-stream (op &optional arg1 &rest rest)
  (selectq op
    (:tyo (push arg1 the-list))
    (:which-operations '(:tyo))
    (otherwise (stream-default-handler
                 #'list-output-stream op arg1 rest))))
```

As an output stream, the stream is required to support :tyo directly, and to support the other standard output operations (like :string-out) via the default handler. The default handler is invoked by calling the function **stream-default-handler**, with arguments of the stream, the operation, the first argument, and the rest arg.

Here's a complementary input stream, which reads its characters from a list.

```
(defvar the-list)

(defvar untyied-char nil)

(defun list-input-stream (op &optional arg1 &rest rest)
  (selectq op
    (:tyi (cond (untyied-char (prog1 untyied-char
                                     (setq untyied-char nil)))
                ((null the-list) (and arg1 (error arg1)))
                (t (pop the-list))))
    (:untyi (setq untyied-char arg1))
    (:which-operations '(:tyi :untyi))
    (otherwise (stream-default-handler
                 #'list-input-stream op arg1 rest))))
```

Note that `:untyi` must be supported, and that the stream must check for having reached the end of the information, and do the right thing with the argument to the `:tyi` operation.

6.2 Accessing Files and Directories

Some of the information in this section will make more sense after reading section 3, Pathnames.

6.2.1 Open, and Other Functions for Operating on Files (10, 10.1, Volume 5)

All reading from and writing to files is done through streams. To access a file you must have an open stream to that file. The fundamental way to obtain an open stream to a file, whether for reading or writing, is with the **open** function.

open *pathname* &rest *options*

Returns a stream connected to the specified file. *pathname* may be anything acceptable to **fs:parse-pathname**, generally either a string or an actual pathname object. *options* is a set of alternating keywords and values, controlling such attributes of the stream as whether it is for input or output, and how many bits there are per "character." Here are some of the more frequently used option keywords:

:direction

usually either **:input**, to read from an existing file, or **:output**, to write a new file.

:byte-size
the number of bits per byte; each **:tyo** or **:tyi** operation will deal with this many bits of the file. Note that this need not agree with what the host computer thinks the byte-size for the file is.

:if-exists (*Warning: not fully supported by Version 8 UNIX.*)
specifies what to do if the `:direction` is `:output` and a file with the desired name already exists. Some of the possibilities are to signal an error, overwrite the old file, append to the end of the old file, or write a file with a unique version number.

Most programs do not call **open** directly. They more commonly use the **with-open-file** macro, which makes use of an **unwind-protect** to guarantee that the stream will be closed when you're done with it. If you call **open** directly, you should also use an **unwind-protect** to make sure the stream gets closed, because leaving around lots of open streams can create problems.

with-open-file (*stream pathname options...*) *body...*
Evaluates the *body* forms with the variable *stream* bound to a stream open for reading or writing to the file specified by *pathname*. *pathname* and *options* are interpreted as in **open**. When control leaves the body, either normally or abnormally, the file is closed. If a new output file is being written, and control leaves abnormally (*i.e.*, because of an error or a **throw**), the file is aborted.

So if I wanted to write a new text file in my directory on the UNIX host sola, which contained only the string "Wow. I'm on a disk!" (without the double quotes), I would evaluate:

```
(with-open-file (str "s://usr//hjb//yippee" :direction :output)
  (send str :string-out "Wow.  I'm on a disk"))
```

Or if I wanted to see how many characters into a certain file the first "a" occurred,

```
(with-open-file (str "s://usr//hjb//.profile" :direction :input)
  (loop for i from 1
        for char = (send str :tyi)
        when (char-equal char #\a) return i))
```

Here are some more functions for operating on files. They generally accept either a string or a pathname object, and some of them also accept a stream open to the appropriate file.

renamef *file new-name* &optional *(error-p t)*

Changes the name of the file. Meta-X Rename File in the editor uses this function. If an error occurs and *error-p* is non-nil, the error is signaled. If there's an error and *error-p* is nil, the error object is returned. See the documentation for details on what happens with wildcard names and links.

deletef *file* &optional *(error-p t)*

The specified file is deleted. *error-p* is as in **renamef**.

fs:file-properties *pathname* &optional *(error-p t)*

Returns a disembodied property list describing the file. The car of the list is a pathname for the file's truename, the rest is alternating indicators and values. See the documentation for **fs:directory-list** for a list of the possible indicators.

fs:change-file-properties *pathname error-p* &rest *properties*

The *properties* arguments are alternating keywords and values. **fs:change-file-properties** alters the attributes of the file accordingly, if possible. (Some properties are not alterable. Which ones are is a property of the host file system.)

viewf *pathname* &optional *(stream* **standard-output***) leader*

Prints the file on *stream*. (Use this just for looking at a file, not for copying it. Its output is not exactly the same as the contents of the file.)

copyf *from-path to-path* &key *(characters* **:default***) (byte-size* **nil***)*
(copy-creation-date **t***) (copy-author* **t***)*
(report-stream **nil***) (create-directories* **:query***)*

Copies one file to another. M-x Copy File in the editor uses this function. See the documentation for the details of merging, wildcard names, and links, and for the meanings of the keyword arguments.

probef *pathname*

If the specified file exists, returns a pathname for its truename. Returns nil if the file does not exist.

load *pathname* &optional *pkg nonexistent-ok-flag dont-set-default-p no-msg-p*

Loads the specified file into the Lisp environment. (If it's a text file, **load** calls **readfile**; if it's a binary (compiled) file, **load** calls **fasload**.)

6.2.2 Special Messages for File Streams (10.3, Volume 5)

These are some of the operations handled by streams connected to files, in addition to the general operations described earlier.

:pathname
Returns the pathname that was opened to get this stream. This may differ from the original argument to **open** because parts of the pathname may have been filled in with defaults.

:truename
Returns the pathname of the file actually open on this stream. This may differ from what **:pathname** returns because of links, logical devices, mapping of the "newest" version onto a specific version number, and so on.

:length
Returns the length of the file, in bytes. The number of bits in each byte depends on how the file was opened. (See the **:byte-size** option to **open**.)

:creation-date
Returns the creation-date of the file, expressed in lisp machine "universal time" units (see "Dates and Times," Part VI of Volume 7).

6.2.3 Directories (11.1, Volume 5)

fs:directory-list *pathname* &rest *options*
pathname may be either a string or a pathname object. **fs:directory-list** finds all files matching *pathname* and for each one gets the information that would be returned by **fs:file-properties** for that file. It collects all of these into a list, and adds one element to the beginning of the list with information about the file system as a whole. So the returned list has one more element than the number of files. See the documentation for a description of the options and more details on what information is provided for each file.

fs:complete-pathname *defaults string type version* &rest *options*
string is a partially specified file name. **fs:complete-pathname** looks in the file system on the appropriate host and returns a new, possibly more specific, string. Any unambiguous abbreviations are expanded out in a host-dependent fashion. There are four full pages of documentation attempting to explain how this happens (repeated in 12.7). Help yourself.

6.3 Pathnames

6.3.1 General (12.1, Volume 5)

Just as streams are intended to provide a uniform, device-independent interface between programs and the different kinds of peripherals, *pathnames* are intended to provide a uniform interface between programs and remote file systems. The idea is to free the programmer from having to keep in mind the format for file names on the various remote hosts. With pathnames, you should be able to manipulate files on a file server without knowing anything about that server's syntax for file names.

All pathnames are instances of some flavor, and all the pathname flavors are built on the flavor **fs:pathname**. Each pathname has six components which correspond to different parts of a file name. The mapping of the components into the parts of the file names is done by the pathname software, and is specific to each kind of host the software knows about.

The six components of a pathname are the *host*, the *device*, the *directory*, the *name*, the *type*, and the *version*. So, for example, the pathname corresponding to the file /usr/hjb/mbox on the UNIX host sola is an instance of flavor **fs:unix-pathname** which prints as `#<UNIX-PATHNAME "S: //usr//hjb//mbox">`. It has a host of sola, a directory of /usr/hjb, a name of mbox, and a value of `:unspecific` for device, type and version. The pathname corresponding to the file >sys>site>notice.text.8 on the lisp machine glengarioch has a host of glengarioch, a directory of >sys>site>, a name of notice, a type of text, a version of 8, and `:unspecific` again for device.

A pathname need not refer to a specific file. `#<UNIX-PATHNAME "S: ">` is a perfectly legitimate pathname, even though it specifies only a host and nothing else.

The conversion of a string into a pathname is usually done by the function **fs:parse-pathname**. The first thing it has to do is determine the host, since the method for parsing the rest of the components depends on which host it is. If there are any colons in the input string, everything appearing before the first colon is considered to be the name of the host. Parsing of the remainder proceeds according to the type of the host, and its own syntax for file names. (If there are no colons, some default value is used for the host — every pathname must have a host.)

Of course, there's no need to go through strings (and worry about the remote host's file name syntax) at all. One of the selling points of pathnames is precisely that you shouldn't need to do so. Accordingly, one may construct pathnames in this manner:

```
(fs:make-pathname :host "s" :directory '("USR" "HJB")
                  :name "MBOX" :type :unspecific)
```

which returns the same pathname whose printed representation was shown above.

Pathnames are *interned*, just like symbols, meaning that there is never more than one pathname with the same set of component values. The main reason for maintaining uniqueness among pathnames is that they have property lists, and it's desirable for two pathnames that look the same to have the same property lists.

6.3.2 Component Values (12.1.4, 12.1.5, Volume 5)

The host component is always a host object (an instance of some flavor built on **net:basic-host**). The permissible values for the other components depends to some extent on the type of the host, but there are some general conventions.

The type is always either a string, or one of the symbols `nil`, `:unspecific` or `:wild`. Both `nil` and `:unspecific` denote a missing component. The difference is in what happens during *merging* (see below); `nil` generally means to use the default, and `:unspecific` generally means to keep that component empty. The symbol `:wild` is sometimes used in pathnames given to **fs:directory-list**, and matches all possible values.

The type field gives an indication of what sort of stuff is in the file. Lisp source files, for instance, usually have a type component of "lisp," and compiled lisp code a type component of "bin." Since there are some system-dependent restrictions on how many characters may appear in this field, a *canonical type* mechanism exists to allow processing of file types in a system-independent fashion. I quote: "A canonical type is a system-independent keyword symbol representing the conceptual type of a file. For instance, a Lisp source file on a VMS* system will have a file type of 'LSP,' and one on a UNIX system will have a file type of 'l.' When we ask pathnames of either of these natures for their canonical type, we receive the keyword symbol `:lisp`."

The version is either a number or one of the symbols `nil`, `:unspecific`, `:wild`, `:newest` or `:oldest`. The first three have the same meaning as for type. `:newest` refers to the largest version number that exists when reading a file, or one greater than that number when writing a new file. `:oldest` refers to the smallest version number that exists.

* VMS is a trademark of Digital Equipment Corporation.

The device component may be either `nil` or `:unspecific`, or a string designating some device, for those file systems that support such a notion (VMS, TOPS-20, ITS).

The name component may be `nil`, `:wild` or a string.

The directory component may be `nil` or `:wild` for any type of host. On non-hierarchical file systems, a string is used to specify a particular directory. On hierarchical systems, the directory component (when not `nil` or `:wild`) is a list of *directory level components*. These are themselves usually strings. So the pathname `#<UNIX-PATHNAME "S: //usr//hjb//mbox">` has for its directory component the list `("USR" "HJB")`. The directory level components can also be special symbols, as well as strings. `:root`, for instance, refers to the root directory on the given host. And `:relative` followed by one or more occurrences of `:up` refers to a relative pathname. So the UNIX pathname `#<UNIX-PATHNAME "S: ..//foo//bar">` has a directory component of `(:relative :up "FOO")`. And the lisp machine pathname `#<LMFS-PATHNAME "G:<<x>y>z.lisp">` has a directory component of `(:RELATIVE :UP :UP "X" "Y")`. Other possibilities for directory level components are `:wild` (for any single directory), `:wild-inferiors` (for any number of directory levels), and partially wild strings, like `"FOO*"`.

6.3.3 Case in Pathnames (12.1.7, Volume 5)

Since the various host systems have different conventions as to upper and lower case characters in file names, most pathname functions perform some standardization of case to facilitate manipulating pathnames in a host-independent manner. There are two representations for any given component value, one in *raw* case and one in *interchange* case. Raw case representation, which is used internally for the instance variables of pathnames, corresponds exactly to what would be sent to the remote machine to find the file corresponding to the pathname. Interchange case is the standardized form, and is what you get if you ask a pathname for its component values. It's also what functions like **fs:make-pathname** expect (unless you specify that you mean raw case).

The standardization is simple. Each host is classified as to whether its preferred case, or system default case, is upper or lower. Any raw component which is in the preferred case for its host has an upper case interchange form. A raw component which is in the non-preferred case has a lower case interchange form. A raw component in mixed case has an identical (mixed case) interchange form. Since UNIX hosts are classified as having lower case for the system default, this means that the raw forms are case-inverted to get the interchange forms, and *vice versa*.

The messages for accessing and setting the component values of pathnames assume that you want to see or set the interchange form, unless you explicitly specify raw case.

6.3.4 Defaults and Merging (12.2, Volume 5)

In most situations where the user is expected to type in a pathname, some *default pathname* is displayed, from which the values of components not specified by the user may be taken. Most programs maintain their own default pathnames, containing component values that would be reasonable in the particular context. For programs which really have no idea of what sort of pathname to expect, there is a set of *default defaults*.

The pathname provided by the user (actually, the pathname constructed by **fs:parse-pathname** from the string provided by the user) and the default pathname are then *merged* by the function **fs:merge-pathnames**. The details are a little messy, but the basic idea is that components which aren't specified in the user's pathname are taken from the default.

6.3.5 Pathname Functions and Messages (12.7, 12.8, Volume 5)

We've already seen three of the most important pathname functions: **fs:parse-pathname, fs:merge-pathnames, and fs:complete-pathname**. Here are some more.

fs:make-pathname &rest *options*

The *options* are alternating keywords and values, specifying the components of the pathname. Missing components default to `nil`, except the host, which is required. Options allowed are `:host`, `:device`, `:directory`, `:name`, `:type`, `:version`, `:raw-device`, `:raw-directory`, `:raw-name`, `:raw-type` and `:canonical-type`. So the device, directory, name and type may be given in either interchange case or raw case, and the type may also be given in canonical form.

fs:define-canonical-type *canonical-type default* &body *specs*

This defines a new canonical type. *canonical-type* is the symbol for the new type, the body is a list of specs giving the surface type corresponding to this canonical type for various hosts. *default* is the surface type for any hosts not mentioned in the body. Here is how the `:lisp` canonical type is defined:

```
(fs:define-canonical-type :lisp "LISP"
  ((:tops-20 :tenex) "LISP" "LSP")
```

```
        (:unix "L" "LISP")
        (:vms "LSP"))
```

The **:host** message to pathnames returns the host component, which will always be an instance of some flavor built on **net:basic-host**. The messages **:device, :directory, :name,** and **:type** return the corresponding component value, with any strings given in interchange case. The messages **:raw-device, :raw-directory, :raw-name,** and **:raw-type** are similar, but use raw case for all strings. The **:version** message returns the version (case is not an issue since versions are never strings). The **:canonical-type** message returns two values; together they indicate the type component of the pathname, and what canonical type — if any — it corresponds to. (See the documentation for details.)

The messages **:new-device, :new-directory, :new-name,** and **:new-type** all take one argument and return a new pathname which is just like the one that received the message except that the value of the specified component will be changed. The argument is interpreted as being in interchange case. You can guess what **:new-raw-device, :new-raw-directory, :new-raw-name** and **:new-raw-type** do. **:new-version** and **:new-canonical-type** also do the obvious thing, and have no "raw" form for the obvious reasons.

:new-pathname allows wholesale replacement of component values; its arguments are alternating keywords and values, with the same keywords accepted as by **fs:make-pathname**.

There are a set of messages for getting strings that describe the pathname. The returned strings come in different forms for different purposes. **:string-for-printing** returns the string that you see inside the printed representation of a pathname. **:string-for-host** shows the file name (not including the host) the way the host file system likes to see it. There are several others.

:get, :putprop, :remprop and **:plist** all do the obvious thing with the pathname's property-list. Keep in mind the distinction between the pathname's property-list and the list returned by **fs:file-properties**, or the **:properties** message to pathnames. The latter are the properties of a file, and require accessing the host's file system. The former are the properties of a pathname, a lisp object which may not even correspond to any files.

6.3.6 Logical Pathnames (12.9.10, Volume 5)

There are some pathnames which don't correspond to any particular file server, but rather to files on a *logical** *host*. The logical host may then be mapped onto any physical host, thus defining a translation from logical pathnames to physical pathnames. This feature improves transportability of code. Take the lisp machine system software as an example. Every lisp machine site keeps the source code on a different computer. But there are many functions that want to be able to find these files, no matter what site they're running at. The solution is to use logical pathnames: all the system software is in files on the logical host "sys." Each site gives the "sys" host an appropriate physical host, and then it works just fine to open a file with a name like "sys: io; pathnm.lisp," which happens to be the file containing the pathname code. At my site that corresponds to the file ">sys-6>io>pathnm.lisp" on the lisp machine laphroaig.

The function **fs:set-logical-pathname-host** defines the mapping of file names from a logical host to the corresponding physical host. The call to **fs:set-logical-pathname-host** is supposed to be placed in the file "sys: site; *host*.translations." Then if you call the function **fs:make-logical-pathname-host** with an argument of the host name, it will look for and load the appropriate file, thus evaluating the **fs:set-logical-pathname-host**. The format of the arguments to **fs:set-logical-pathname-host** is best explained by example. This is an abridged version of the contents of "sys: site; kwc.translations," which defines the "kwc" logical host:

```
(fs:set-logical-pathname-host
  "kwc"
  :physical-host "Sola"
  :translations
    '(("distribution;" "//lispm//kwc//rel6//distribution//")
      ("distribution;*;" "//lispm//kwc//rel6//distribution//*//")
      ("distribution;*;*;" "//lispm//kwc//rel6//distribution//*//*//")
      ("*;" "//lispm//kwc//rel6//*//"))
  :rules
    '((:unix
        ("kwc:**;public-systems.*.*" :new-pathname :name "PUBSYS")
        ("kwc:**;input-editor.*.*" :new-pathname :name "INPUT-ED")
        ("kwc:**;comtab-example.*.*" :new-pathname :name "COMTAB-EX")
        ("kwc:**;call-mini-buffer.*.*" :new-pathname :name "CALL-MINI")
        ("kwc:**;moving-icons.*.*" :new-pathname :name "MVG-ICONS"))))
```

As you can see, the mapping is done on a directory-by-directory basis. Wild-cards are allowed. For instance, files in the directory "kwc: distribution; new-class" are mapped to the directory "/lispm/kwc/rel6/distribution/new-class/" by the second entry in the `:translations` argument. The `:rules` argument allows us to

* See hacker's definition at end of chapter.

specify additional transformations to be carried out in special cases. Since our version of UNIX doesn't allow filenames longer than 14 characters, we take advantage of this facility to define shortened names for the UNIX physical filenames corresponding to logical pathnames with long names.

Given a pathname for some logical host, the mapping to physical pathname is carried out by sending the logical pathname the **:translated-pathname** message. An earlier version of the text for this chapter is in a file whose logical pathname is `#<LOGICAL-PATHNAME "KWC: DISTRIBUTION; NEW-CLASS; STREAMS.TALK">`. When that pathname is sent the `:translated-pathname` message, it returns `#<UNIX-PATHNAME "S://lispm//kwc//rel6//distribution//new-class//streams.talk">`.

6.4 Fun and Games

From *The Hacker's Dictionary*, Guy L. Steele, Jr., *et al*:

LOGICAL *adjective.*

> Conventional; assumed for the sake of exposition or convenience; not the actual thing but in some sense equivalent to it; not necessarily corresponding to reality.
>
> Example: If a person who had long held a certain post (for example, Les Earnest at Stanford) left and was replaced, the replacement would for a while be known as the "logical Les Earnest." Pepsi might be referred to as "logical Coke" (or vice versa).
>
> At Stanford, "logical" compass directions denote a coordinate system in which "logical north" is toward San Francisco, "logical south" is toward San Jose, "logical west" is toward the ocean, and "logical east" is away from the ocean — even though logical north varies between physical (true) north near San Francisco and physical west near San Jose. The best rule of thumb here is that El Camino Real by definition always runs logical north-and-south. In giving directions, one might way, "To get to Rincon Tarasco Restaurant, get onto EL CAMINO BIGNUM going logical north." Using the word "logical" helps to prevent the recipient from worrying about the fact that the sun is setting almost directly in front of him as he travels "north."
>
> A similar situation exists at MIT. Route 128 (famous for the electronics industries that have grown up along it) is a three-quarters circle surrounding Boston at a radius of ten miles, terminating at the coast line at each end. It

would be most precise to describe the two directions along this highway as being "clockwise" and "counterclockwise," but the road signs all say "north" and "south," respectively. A hacker would describe these directions as "logical north" and "logical south," to indicate that they are conventional directions not corresponding to the usual convention for those words. (If you went logical south along the entire length of Route 128, you would start out going northwest, curve around to the south, and finish headed due east!)

6.5 Problem Set #6

Questions

1. A. There is a function named **print-disk-label** that, when called with no arguments, prints on the screen a listing of the contents of the fep file system. Find a way to print this listing to a file instead. (Hint: check out the optional arguments to **print-disk-label**, and use **with-open-file**.)

   B. There's another function named **si:print-login-history** that prints a list of everyone who has logged in to the local machine since it was cold-booted (and the contents of the login history when the world load was made). This one has no optional argument for what stream to do the printing on — it always prints to **standard-output**. How can you get it to print the listing to a file? (Hint: make **standard-output** point to a file.)

2. Suppose there is a file whose contents are numbers in the range $-2{,}147{,}483{,}648 \leqslant n \leqslant 2{,}147{,}483{,}647$ (32-bit integers). How can we read the file into an array of fixnums? It'd be convenient to open a 32-bit stream to the file and just do `:tyi`'s or a `:string-in`, but most file servers won't allow a 32-bit stream. We'll have to use a 16-bit stream. One strategy is to read two 16-bit bytes at a time and build a 32-bit number by shifting one number 16 bits and adding them together. This will work, but it's awfully slow. Can you think of anything better? (Hint: think about displaced arrays of different types.)

Solutions

1. A.
```
(with-open-file (str "s://usr//hjb//disk-label"
                     :direction :output)
  (print-disk-label si:*boot-unit* str))
```
   B.
```
(with-open-file (standard-output "s://usr//hjb//logins"
                                 :direction :output)
  (si:print-login-history))
```

2. The slow way:

```
(defun foo (file &optional array)
  (with-open-file (str file :characters nil :byte-size 16.)
    (or array (setq array
                    (make-array (// (send str :length) 2))))
    (loop for i from 0
          for c1 = (send str :tyi)
          for c2 = (send str :tyi)
          while c1
          do (setf (aref array i) (+ c1 (lsh c2 16))))
    array))
```

The fast way:

```
(defun bar (file &optional array32 array16)
  (with-open-file (str file :characters nil :byte-size 16.)
    (or array32
        (setq array32 (make-array (// (send str :length) 2)
                                  :initial-value 0)))
    (send str :string-in nil
          (or array16 (make-array (* 2 (array-length array32))
                                  :type 'art-16b
                                  :displaced-to array32)))
    array32))
```

Chapter 7

THE TREE EXAMPLE

This chapter is very much like the graph example two chapters back — a later section contains a code listing, and this one describes some of the new features* of the code. Much of the code was copied directly from the graph example (with "graph" changed to "tree"). The most interesting of the new parts have to do with menus.

Once again, if your site has the tape for this book, you can load the code by using the CP command `Load System tree` [or evaluating `(make-system 'tree)`]. Once the code has been read, start the program by evaluating `(send (tv:make-window 'tree-frame) :select)`.

7.1 The Nodes and Arcs

root and ***the-real-root***

These two variables are declared at the very beginning of the file. Rather than keep a list of all the nodes, as "graph" does, we simply keep track of the root of the

* See hacker's definition at end of chapter.

tree and follow the connections from there. The value of ***the-real-root*** is constant throughout the lifetime of a given tree. It changes only when we throw away the tree to start another. ***root*** refers to the node which is displayed at the top center of the window. Initially, this is the same as ***the-real-root***, but you can change it to be any arbitrary node in the tree. That way you can move your window around over a tree too large to be viewed at once.

The **node** defflavor

The connections between nodes (*arcs*, in "graph") are no longer real data objects. Each node now knows directly which other nodes it's connected to, instead of knowing which arcs it's connected to. *children* is a list (possibly empty) of the direct inferiors, and *parent* is the superior (nil for the node which is the value of ***the-real-root***). The x-pos and y-pos instance variables have been thrown out, since nodes no longer have fixed positions. Each parent determines where its children will be drawn. The *space-requirements* instance variable somehow packages up everything a node's parent needs to know about the node and its children in order to determine where to draw it. The first time this information is requested it is recursively calculated, and saved for future requests. The saved info is flushed whenever something happens that would invalidate it (such as a change in the number of children), so that it is recalculated the next time someone asks for it. Consequently, anyone needing the information should use the `:get-space-requirements` method, which calculates if necessary, rather than looking at the instance variable directly.

The `:flush-space-requirement` method

This method is called whenever the space-requirement info has been invalidated. It sets the instance variable to nil, and recurses upward, because anytime a node's space requirements change its parent's also do. It gets called whenever the node's label is changed, and whenever a child is added or removed. (And whenever any of these things happens to one of its descendants.)

Fancier **format** directives

The **format** statement inside the `:print-self` method uses two features you may not have seen before. "`~{...~}`" is an iteration construct, which takes a list as an argument, and repeats the interior of the { }'s until the list elements are exhausted. "`~@[...~]`" checks the next argument, and if it is nil, does nothing. If it is non-nil, the argument is not used up but remains the next one to be processed, and the interior of the [ ]'s is executed.

The effect here is to print a list of all the children (using `~A` so that only their names are printed), with all but the last in the list being followed by "; ". Both `~{` and `~[` exist in several forms, with and without `:` and/or `@`, allowing various kinds of iteration and selection.

Drawing

The methods `:draw-self-and-children` and `:get-space-requirements` have all the smarts. It's complicated, but I don't think it's particularly interesting. I won't go into it here since there's little of general value. Do feel free, however, to look through the code on your own.

7.2 The Windows and the Mouse

As with "graph," the first half of the file is adequate if you're willing to type awkward forms to a lisp listener. The second half provides a better user interface, mainly using the mouse.

The **tree-frame** defflavor

This time we have three panes instead of two. The new one is a *command menu*. Many system utilities based on frames have a command menu pane, including Peek (Select P), Zmail (Select M), and File System Maintenance (Select F). Command menus differ from other menus in that they stay exposed indefinitely, becoming active only when you move the mouse over them, and in that they don't themselves produce any action when you choose an item, but simply stuff a blip into somebody's io-buffer. It's up to whoever reads from the io-buffer to do something with the blip (often sending it back to the command menu with an `:execute` message). There are more details on how menus work later in the chapter.

Shared io-buffers

For the process running in the tree pane to see the blips from the command menu, the two windows must share one io-buffer. The `:after :init` method on tree-frame arranges this. I could have built tree-frame on **tv:bordered-constraint-frame-with-shared-io-buffer** instead of plain **tv:bordered-constraint-frame**, but then the lisp pane would also share the one io-buffer, and that wouldn't work. (Some input intended for the tree-pane's process would be read by the lisp-pane's process, and *vice versa*.)

Tree-window's `:main-loop`

There are now two kinds of blip to watch for: the familiar `:typeout-execute` blips from the mouse-sensitive items, and new `:menu` blips, from the command menu. For now, all you need to understand about the action taken for `:menu` blips is that the command-menu itself is sent an `:execute` message with an argument of the menu item that was chosen.

Tree-window's `:refresh`

Drawing the tree is accomplished by sending `:draw-self-and-children` to the current ***root***, which will recursively send the same message to its descendants. The initial arguments put the ***root*** at the top-center of the window.

The menu-item-list

Now we get into menus. First off, I should mention that the documentation on menus — Part III of volume 7 — is not too bad. (In fact, much of this stuff I had actually never messed with until the day before writing the first version of this chapter. I just read the documentation and did it.) The basic idea is that menus are special kinds of windows that maintain a list of items to choose from, and do something appropriate if you click on one of the items. The list of items is kept in the instance variable *item-list*; once a menu has the right item-list, the `:choose` method does the rest: it exposes the window, waits for you to click on some item (the `:mouse-buttons` method tells it when that has happened by setting the *chosen-item* instance variable), and sends itself the `:execute` message with an argument of the chosen item. The `:execute` message does something appropriate, depending on the type of the item.

And now for the format of the items on the item-list, and what it means to "do something appropriate" (the task of the `:execute` message). The simplest kind of item is just a string or a symbol. The string or symbol is displayed in the menu as itself, and executing such an item just means to return it. The item may also be a list (or dotted-pair) of two elements; the first is the symbol or string to be displayed, the second is what is returned by execution of the item. The most general kind of item is a list of three or more elements. The first is what to display in the menu, the second is a keyword for the type of this item, the third is an arbitrary argument whose interpretation depends on the type of the item, and the rest of the list is alternating pairs of modifier keywords and values. The keyword in position two may be any of: `:VALUE`, `:EVAL`, `:FUNCALL`, `:FUNCALL-WITH-SELF`, `:NO-SELECT`, `:WINDOW-OP`, `:KBD`, `:MENU`, or `:BUTTONS`. The first three are the most commonly used. `:VALUE` means to return the argument (the third element of the list), `:FUNCALL` means to funcall the argument (presumably the name of a function) and return its return value, and `:EVAL` means to evaluate the argument (presumably a lisp form) and return its return value. The only defined modifier keywords are `:FONT` and `:DOCUMENTATION`. `:FONT` specifies which font should be used to display this item in the menu, `:DOCUMENTATION` is what appears in the who-line when the mouse is over this item.

If you look at the ***tree-command-menu-item-list***, you'll see that all three items use the general form. Two of them are of the `:FUNCALL` type and one is of the `:EVAL` type.

tv:menu-choose

The easiest way to use menus is to call the function **tv:menu-choose** with an argument of a suitable item-list. This function will allocate and expose a menu, set its item-list instance variable to be the argument you supplied, and send it the `:choose` message. Its `:choose` method then waits for you to click on an item, and sends the menu `:execute` of the chosen item. If you look at the `:mouse-click` method for tree-window, you'll see that clicking right anywhere except over a mouse-sensitive item does just that. It calls **tv:menu-choose** with an argument of ***tree-command-menu-item-list***. (The use of **process-run-function** is necessary because the `:mouse-click` method runs inside the mouse process — without it the mouse process would be hung until tv:menu-choose returned, but tv:menu-choose would never return because the mouse process is hung.)

The command menu revisited

The command menu pane uses exactly the same item-list, as you can see from looking at the tree-frame defflavor, in the `:panes` section. But as I said earlier, command menus work differently; you don't send them the `:choose` message. They stay exposed indefinitely, and when you click on one of their items they just stuff a blip into their own io-buffer. In our case, that means the tree pane's io-buffer. When the `:main-loop` finds it there, it sends the `:execute` message back to the command menu (with most menus the `:choose` method does this for you), and the chosen item is executed normally.

7.3 Fun and Games

From *The Hacker's Dictionary*, Guy L. Steele, Jr., *et al*:

FEATURE *noun.*

1. An intended property or behavior (as of a program). Whether it is good is immaterial.

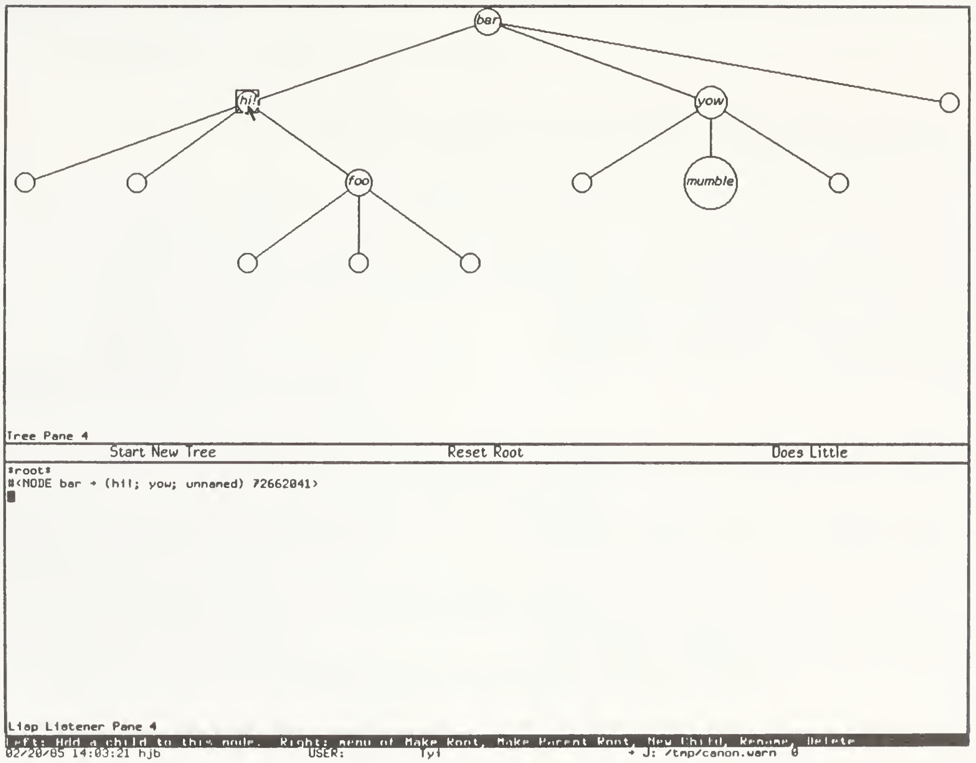
2. A good property or behavior (as of a program). Whether it was intended is immaterial.
3. A surprising property or behavior; in particular, one that is purposely inconsistent because it works better that way. For example, in the EMACS text editor, the "transpose characters" command will exchange the two characters on either side of the cursor on the screen, *except* when the cursor is at the end of a line; in that case, the two characters before the cursor are exchanged. While this behavior is perhaps surprising, and certainly inconsistent, it has been found through extensive experimentation to be what most users want. The inconsistency is therefore a feature and not a BUG.
4. A property or behavior that is gratuitous or unnecessary, though perhaps impressive or cute. For example, one feature of the MACLISP language is the ability to print numbers as Roman numerals. See BELLS AND WHISTLES.
5. A property or behavior that was put in to help someone else but that happens to be in your way. A standard joke is that a bug can be turned into a feature simply by documenting it (then theoretically no one can complain about it because it's in the manual), or even by simply declaring it to be good. "That's not a bug; it's a feature!"

   The following list covers the spectrum of terms used to rate programs or portions thereof (except for the first two, which tend to be applied more to hardware or to the SYSTEM, but are included for completeness):

CRASH	BUG	CROCK	WIN
STOPPAGE	LOSS	KLUGE	FEATURE
BRAIN DAMAGE	MISFEATURE	HACK	PERFECTION

   The last is never actually attained.

7.4 The Program



```
*root*
#<NODE bar → (hi!; yow; unnamed) 72662041>
```

```
Left: Add a child to this node.  Right: menu of Make Root, Make Parent Root, New Child, Rename, Delete
02/20/85 14:03:21 hjb          USER:        Tyi              → J: /tmp/canon.warn  0
```

```
;;; -*- Syntax: Zetalisp; Mode: LISP; Package: USER; Base: 10 -*-

;;; display of trees, in the same style as the earlier "graph" example

(defvar *the-real-root* nil)                    ; the "node" instance at the root of the tree
(defvar *root* nil)                             ; the node currently shown in root position

(defvar *tree-window* nil)                      ; the window in which the tree is displayed

(defvar *minimum-node-radius* 10)               ; the size (in pixels) of nodes with no label
(defvar *node-margin-size* 3)                   ; pixels between label and perimeter of node

(defvar *vertical-spacing* 30)                  ; number of pixels between rows in the tree
(defvar *horizontal-spacing* 100)               ; pixels between adjacent nodes within a row


(defflavor node
        ((children nil)                         ; list of other nodes (inferiors of this one)
         (parent nil)                           ; this node's parent (nil if we're the root)
         (radius *minimum-node-radius*)         ; in pixels (dependent on length of label)
         (label nil)                            ; a string, to be displayed in the node
         (space-requirements nil)               ; space needed by this node and its children
         ;; [This is a list of five elements:  the total horizontal space needed by this node
         ;; together with its descendents; a list of the total horizontal space needed by each
         ;; of the children; the space needed by all the children together (the sum of the
         ;; previous list plus padding - this differs from the total space if this node needs
         ;; more space than all its children put together, which happens if there are no
         ;; children, or if this node is big and it has one small child); a list of the radii
         ;; of each of the children (for drawing the connecting lines); and the max of the
         ;; radii of all descendants, including self (for determining vertical spacing)]
         )
        ()
  (:settable-instance-variables label parent)
  (:gettable-instance-variables children radius))
```

```
(defun-method reset-radius node ()
  (setq radius
        (if label
            (max *minimum-node-radius*
                 (+ *node-margin-size*
                    (// (send *tree-window* :string-length label) 2)))
            *minimum-node-radius*)))

(defmethod (node :init) (&rest ignore)
  ;; set the radius according to the length of the label (in the current font)
  (cond ((not label))
        ((not *tree-window*)
         (tv:notify nil
                    "Warning:  node ~S created while there is no tree window; radius not set"
                    self))
        (t (reset-radius))))

(defmethod (node :after :set-label) (ignore)
  (and *tree-window* (reset-radius))
  (send self :flush-space-requirements))

(defmethod (node :flush-space-requirements) nil
  (setq space-requirements nil)
  (and parent (send parent :flush-space-requirements)))

(defmethod (node :print-self) (stream ignore slashify-p)
  (let ((name (or label "unnamed")))
    (if slashify-p
        (SI:PRINTING-RANDOM-OBJECT (SELF STREAM :TYPEP)
          (format stream "~A → (~{~A~@[; ~]~})" name children))
        (princ name stream))))
```

```
(defmethod (node :draw-self) (window x y)
  (send window :draw-circle x y radius)
  (and label
       (send window :display-centered-string label
             (- x radius) (+ x radius) (- y (// (send window :line-height) 2))))
  (send window :send-if-handles
        :primitive-item :node self (- x radius) (- y radius) (+ x radius) (+ y radius)))

(defmethod (node :draw-self-and-children) (window x y &optional y-inc)
  (send self :draw-self window x y)
  (when children
    (destructuring-bind (nil individual-requirements inferiors-total radii . max-radius)
                        (send self :get-space-requirements)
      (memoizef y-inc (+ *vertical-spacing* (* 2 max-radius)))
      (loop with next-y = (+ y y-inc)
            for c in children
            for req in individual-requirements
            for r in radii
            for left = (- x (// inferiors-total 2)) then (+ right *horizontal-spacing*)
            for right = (+ left req)
            for new-x = (// (+ left right) 2)
            do (send c :draw-self-and-children window new-x next-y y-inc)
               (connect-nodes window radius x y r new-x next-y)))))
```

```
(defmethod (node :get-space-requirements) nil
  (memoizef space-requirements
    (let ((own-space (* 2 radius)))
      (if (null children)
          (list* own-space nil nil nil radius)
          (loop for c in children
                for (space nil nil nil . inf-max-rad) = (send c :get-space-requirements)
                for hor-padding = 0 then *horizontal-spacing*
                for rad = (send c :radius)
                collect space into reqs
                sum (+ space hor-padding) into total
                collect rad into rads
                maximize inf-max-rad into max-rad
                finally (return (list* (max total own-space)
                                       reqs total rads (max max-rad radius))))))))

(defmethod (node :add-child) (child)
  (push child children)
  (send child :set-parent self)
  (send self :flush-space-requirements))

(defmethod (node :remove-child) (child)
  (setq children (delq child children))
  (send child :set-parent nil)
  (send self :flush-space-requirements))

(defmethod (node :delete) nil
  (and parent (send parent :remove-child self)))

(defun connect-nodes (window r1 xpos1 ypos1 r2 xpos2 ypos2)
  (multiple-value-bind (x1 y1 x2 y2)
      (find-edges-of-nodes r1 xpos1 ypos1 r2 xpos2 ypos2)
    (send window :draw-line x1 y1 x2 y2)))
```

```
(defun find-edges-of-nodes (r1 xpos1 ypos1 r2 xpos2 ypos2)
  (let* ((dx (- xpos2 xpos1))
         (dy (- ypos2 ypos1))
         (length (isqrt (+ (* dx dx) (* dy dy)))))
    (values (+ xpos1 (// (* dx r1) length))
            (+ ypos1 (// (* dy r1) length))
            (- xpos2 (// (* dx r2) length))
            (- ypos2 (// (* dy r2) length)))))


(defflavor tree-frame ()
           (tv:bordered-constraint-frame)
  (:default-init-plist
            :selected-pane 'lisp
            :panes `((tree tree-pane)
                     (menu tv:command-menu-pane :item-list ,*tree-command-menu-item-list*)
                     (lisp tv:lisp-listener-pane))
            :configurations '((main (:layout (main :column tree menu lisp))
                                    (:sizes (main (tree 0.6)
                                                  :then (menu :ask :pane-size)
                                                  :then (lisp :even)))))))

(defmethod (tree-frame :after :init) (&rest ignore)
  (send self :send-pane 'menu :set-io-buffer
        (send self :send-pane 'tree :io-buffer)))

;;; This makes our frame accessible via Select-L, because the Select key now thinks
;;; it's a lisp-listener instead of a tree-frame.  (The method we are overriding is
;;; on basic-frame and would return the tree-frame itself.)
(defmethod (tree-frame :alias-for-selected-windows) ()
  tv:selected-pane)
```

```
;;; Don't ask.  The sneaky thing we just did to make the Select key see the lisp pane at top
;;; level confuses the part of the window system which does things like make the list of
;;; windows for the select option of the system menu.  It would have the lisp pane listed
;;; twice (once for itself and once for the tree-frame).  This fixes it.  (The method we
;;; are overriding is on essential-window.)
(defmethod (tree-frame :selectable-windows) ()
  `((,(send tv:selected-pane :name-for-selection) ,tv:selected-pane)))

(defflavor tree-pane ()
           (tv:pane-no-mouse-select-mixin tree-window))

(defflavor tree-window ()
           (tv:process-mixin
            tv:basic-mouse-sensitive-items
            tv:window)
  (:default-init-plist :process '(tree-window-top-level-function)
                       :item-type-alist *tree-item-type-alist*
                       :blinker-p nil
                       :font-map '(fonts:hl12i)))

(defmethod (tree-window :after :init) (&rest ignore)
  (setq *tree-window* self))

(defun tree-window-top-level-function (window)
  (send window :main-loop))

(defmethod (tree-window :main-loop) nil
  (loop for blip = (send self :any-tyi)
        do (selectq (first blip)
             ;; format of blips from ms items is (:typeout-execute operation ms-item)
             (:typeout-execute (funcall (second blip) (third blip) self))
             ;; format of blips from c'mand menu is (:menu chosen-item button-mask menu-pane)
             (:menu (send (fourth blip) :execute (second blip))))))
```

```
(defmethod (tree-window :after :refresh) (&rest ignore)
  (let ((root *root*))               ; locals are faster - worth it for repeated references
    (and root (send root :draw-self-and-children
                         self (// (tv:sheet-inside-width) 2) (send root :radius)))))

(defvar *tree-command-menu-item-list*
        '(("Start New Tree" :funcall start-new-tree :documentation
           "Make a new tree with a single node, flushing the current tree (if any).")
          ("Reset Root" :funcall reset-root
           :documentation "Bring the original root of this tree into root position.")
          ("Does Little" :eval (tv:notify nil "Somebody clicked on me!")
           :documentation "Demonstrate /"Eval/" item type.")))

(defun start-new-tree ()
  (setq *the-real-root* (setq *root* (make-instance 'node)))
  (send *tree-window* :refresh))

(defun reset-root ()
  (setq *root* *the-real-root*)
  (send *tree-window* :refresh))

;;; this is redundant, duplicating exactly the functionality of the command menu pane
(defmethod (tree-window :mouse-click) (buttons x y)
  (and (not (send SELF ':MOUSE-SENSITIVE-ITEM X Y))     ; do nothing when over m-s items
       (= buttons #\mouse-r-1)
       (progn (process-run-function "Menu" #'tv:menu-choose *tree-command-menu-item-list*)
              t)))

(defvar *tree-item-type-alist* nil)

(tv:add-typeout-item-type *tree-item-type-alist* :node "Make Parent Root"
                          mouse-make-parent-root nil
                          "Redisplay the tree with this node's parent in root position.")
```

```
(tv:add-typeout-item-type *tree-item-type-alist* :node "Make Root" mouse-make-node-root nil
                          "Redisplay the tree with this node in root position.")

(tv:add-typeout-item-type *tree-item-type-alist* :node "Delete" mouse-delete-node nil
                          "Splice this node out of the tree.")

(tv:add-typeout-item-type *tree-item-type-alist* :node "Rename" mouse-rename-node nil
                          "Provide a new label for this node.")

(tv:add-typeout-item-type *tree-item-type-alist* :node "New Child" mouse-new-child t
                          "Add a child to this node.")

(defun mouse-new-child (node window)
  (send node :add-child (make-instance 'node))
  (send window :refresh))

(defun mouse-rename-node (node window)
  (send node :set-label (tv:get-line-from-keyboard "Enter a label for the node"
                                                   tv:mouse-sheet #'readline-or-nil))
  (send window :refresh))

(defun mouse-delete-node (node window)
  (send node :delete)
  (send window :refresh))

(defun mouse-make-node-root (node window)
  (setq *root* node)
  (send window :refresh))

(defun mouse-make-parent-root (node window)
  (let ((parent (send node :parent)))
    (if (not parent) (beep)
        (setq *root* parent)
        (send window :refresh))))
```

```
;;; Here is the definition of memoizef.  Since it's a macro, it MUST be loaded
;;; BEFORE the rest of this file is compiled to get correct compiled code.  I've
;;; put this definition in a separate file, and arranged for that file to be
;;; loaded first when you make the "tree" system.
;
; (defmacro memoizef (loc &body body)
;   "Use loc to memoize the result of body.
; If loc is non-nil, then it holds the previous result of computing the body,
; and it can be used instead.  If loc is nil, then compute the body and stuff
; the result in loc and then return the result."
;   (let ((value (gensym)))
;     `(or ,loc
;          (let ((,value (progn .,body)))
;            (setf ,loc ,value)
;            ,value))))
```

7.5 Problem Set #7

Questions

1. Write a function which puts up a menu with five choices: "Truth," "Falsehood," "Confusion," "Panic," and "Mega-panic!." Clicking on "Truth" returns the string "Truth;" clicking on "Falsehood" also returns the string "Truth;" clicking on "Confusion" randomly returns one of the strings "Truth" or "Falsehood;" clicking on "Panic" enters the Fep; clicking on "Mega-panic!" enters the Fep and does a cold boot.

2. Use the tree code to display flavor component trees. Ignore "included-flavors" if you like.

3. You may have noticed that in the editor (and in some other contexts), putting the mouse in the top-right or bottom-right corners causes the mouse blinker to change to a squat vertical arrow, and bumping it against the top or bottom edge of the window causes scrolling. This behavior is called "flashy-scrolling," and is controlled by the flavor **tv:flashy-scrolling-mixin**. Add this mixin to the tree window and write the necessary methods so that bumping the mouse against the top-right corner will "scroll," *i.e.*, make the parent of the current root be the root.

4. There's an awful lot of duplication in the graph and tree examples. The right thing to do would have been to take out the common portions, and build graph and tree on top of them. Re-implement graph and tree in that fashion. (It sounds like a lot of work, but I'll bet it won't take more than an hour.)

Hints

1. Your function need simply call **tv:menu-choose** with an argument of an appropriate item list. See section 14.3 of volume 7 for a description of the possible forms for menu items. The last three menu items will have to be of the "general list" form.

2. Constructing a tree which corresponds exactly to the ordered list of flavors used for building combined methods is tricky because of "included-flavors." But it's not hard to make a tree showing the regular components. Given a flavor's name in the form of a symbol, `(si:flavor-depends-on (get` *flavor-name* `'si:flavor))` returns a list of the components. Give the node corresponding to a flavor one child for each of the flavor's components, and recurse on them. To make the tree fit on the screen you'll probably want to switch to a smaller font, and decrease the values of ***vertical-spacing*** and ***horizontal-spacing***.

3. The best way to find out about **tv:flashy-scrolling-mixin** is to read the source code. There isn't very much of it. You'll have to write methods to handle the messages `:scroll-more-above` and `:scroll-more-below` (which should return `t` when it is meaningful to scroll in the given direction and `nil` when it is not), the message `:y-scroll-to`, which should do the actual scrolling, the message `:scroll-bar-p`, which allows or inhibits scrolling, and the message `:handle-mouse-scroll`. I suggest that `:scroll-more-above` return `nil` when ***root*** is **eq** to ***the-real-root***, and `t` otherwise; that `:scroll-more-below` always return `nil`; that `:y-scroll-to` call the function **mouse-make-parent-root** on ***root*** and **self**; that `:scroll-bar-p` always return `t`; and that `:handle-mouse-scroll` always return `nil`. (The `:handle-mouse-scroll` method isn't directly related to flashy-scrolling, but because our `:scroll-bar-p` method returns `t`, the system is going to think that in addition to flashy-scrolling, we have a standard scroll bar in the left margin. So every time the mouse bumps against the left margin, our window will be sent that message. The method needn't do anything, but it had better be defined or we'll get an error. This may sound like a modularity problem, and in fact it is. It appears that someone assumed only windows with scroll bars would use flashy scrolling.) Don't forget to make another tree-frame, since mixing in **tv:flashy-scrolling-mixin** will be an incompatible change to **tree-window**.

4. The basic part might define flavors **node**, **node-window**, **node-pane** and **node-frame**, and methods for them. Graph would then build **graph-node** on top of **node**, and **graph-pane** and **graph-frame** on top of **node-pane** and **node-frame**. Tree would similarly build **tree-node** on top of **node** and **tree-pane** and **tree-**

frame on top of **node-pane** and **node-frame**.

Solutions

1.
```
(defun silly-menu ()
  (tv:menu-choose
    '("Truth"
      ("Falsehood" "Truth")
      ("Confusion" :eval (nth (random 2)
                              '("Falsehood" "Truth")))
      ("Panic" :funcall si:halt)
      ("Mega-panic!" :eval (si:halt (format nil "b~%"))))))
```

2. Here is a crude first pass at making flavor component trees. It ignores included flavors as well as duplication of components.

```
(setq *vertical-spacing* 5)
(setq *horizontal-spacing* 10)

(send *tree-window* :set-font-map '(fonts:tiny))

(defun show-flavor-tree (flavor-name)
  (start-new-tree)
  (send *the-real-root* :set-label
                        (format nil "~S" flavor-name))
  (show-flavor-subtree flavor-name *the-real-root*)
  (send *tree-window* :refresh))

(defun show-flavor-subtree (flavor-name parent)
  (loop for component-flavor
            in (reverse (si:flavor-depends-on
                          (get flavor-name 'si:flavor)))
        for node = (make-instance 'node :label
                         (format nil "~S" component-flavor))
        do (send parent :add-child node)
           (show-flavor-subtree component-flavor node)))
```

We can avoid duplication by keeping a list of all the flavors seen so far, and using a new flavor only if it isn't already on the list. (Modifications in upper case.) If you want try accounting for all the vagaries of "included-flavors," look at the function **si:compose-flavor-inclusion**.

```
(DEFVAR *SEEN-LIST*)
```

```
(defun show-flavor-tree (flavor-name)
  (start-new-tree)
  (send *the-real-root* :set-label
                        (format nil "~S" flavor-name))
  (LET ((*SEEN-LIST* (LIST FLAVOR-NAME)))
    (show-flavor-subtree flavor-name *the-real-root*))
  (REVERSE-ALL-DESCENDANTS *THE-REAL-ROOT*)
  (send *tree-window* :refresh))

(defun show-flavor-subtree (flavor-name parent)
  (loop for component-flavor
            in (si:flavor-depends-on
                 (get flavor-name 'si:flavor))
        UNLESS (MEMQ COMPONENT-FLAVOR *SEEN-LIST*)
          DO (LET ((node (make-instance 'node :label
                           (format nil "~S"
                                       component-flavor))))
               (PUSH COMPONENT-FLAVOR *SEEN-LIST*)
               (send parent :add-child node)
               (show-flavor-subtree component-flavor node))))
```

This function will make nonsense of any existing space requirement info. Fortunately, we can be sure there won't be any, since these nodes were just created and have never been displayed. (We couldn't just reverse the children as we get to them the way we did above because then we'd keep the wrong node when a flavor appears more than once.)

```
(DEFUN REVERSE-ALL-DESCENDANTS (NODE)
  (SEND NODE :EVAL-INSIDE-YOURSELF
            '(SETQ CHILDREN (NREVERSE CHILDREN)))
  (LOOP FOR CHILD IN (SEND NODE :CHILDREN)
        DO (REVERSE-ALL-DESCENDANTS CHILD)))
```

3.
```
(defflavor tree-window ()
           (tv:flashy-scrolling-mixin   ; this was added
            tv:process-mixin
            tv:basic-mouse-sensitive-items
            tv:window)
  (:default-init-plist
     :process '(tree-window-top-level-function)
     :item-type-alist *tree-item-type-alist*
     :blinker-p nil
     :font-map '(fonts:hl12i)))
```

```
(defmethod (tree-window :scroll-more-above) nil
  (neq *root* *the-real-root*))

(defmethod (tree-window :scroll-more-below) nil
  nil)

(defmethod (tree-window :y-scroll-to) (ignore ignore)
  (mouse-make-parent-root *root* self))

(defmethod (tree-window :scroll-bar-p) nil
  t)

(defmethod (tree-window :handle-mouse-scroll) nil
  nil)
```

4. See the file “book: tape; graph&tree.lisp”

Chapter 8

RANDOM USEFUL TOPICS: RESOURCES & SYSTEMS

8.1 Resources

The documentation on resources (chapter 18 of volume 8) is not bad. What follows is primarily a condensation of it.

In cases where a program uses and then discards large objects at a high rate, it can be worthwhile to do the storage management manually, rather than relying on the garbage collector to eventually clean up. The *resource* facility provides a simple way to do so, and is widely used throughout the system software. The Chaosnet code allocates and frees *packets*, which are moderately large, at a very high rate. The window system allocates and frees certain kinds of windows, which are very large, moderately often. Both use resources.

For each resource defined, there is a free list kept of suitable objects. *Allocating* a resource involves checking the list of available objects and returning one if there are any. If not, a new one is created and returned. *Deallocating* a resource involves returning a previously allocated object to the free list. So the storage space occupied by a deallocated object is not really freed in the sense that the garbage collector reclaims free space; it does not become available to be used as part of any newly created lisp object. The original object continues to occupy the storage space, but may itself be reused through being allocated again.

The four functions and macros which together compose the resource facility are **defresource**, for defining a new resource, **allocate-resource**, for allocating an object from a resource, **deallocate-resource**, for freeing an allocated object, and **using-resource**, which temporarily allocates an object and then deallocates it.

A call to **defresource** looks like this:

```
(defresource name parameters
   keyword value
   keyword value
   ... )
```

name should be a symbol, which will be the name of the resource, and which will get a `defresource` property of the data object representing the resource. *parameters* is a (possibly empty) lambda-list of pseudo-arguments which will restrict the objects considered suitable to some subset of those on the free list. For instance, a resource of 2D arrays might have two parameters, the number of rows and the number of columns. When allocating an object from this resource, you could specify how many rows and columns it should have. The free list would be filtered for arrays with the requested dimensions — if all the arrays on the free list had the wrong dimensions, a new one would be created.

There are seven possible keyword options. Only one is required, the **:constructor** option.

:constructor
: The *value* is either a form or the name of a function. It is responsible for making an object, and is used when someone tries to allocate an object and there are no suitable free ones. If *value* is a function, it is given the internal data structure for the resource and any supplied parameters as its arguments. If it is a form, it may access the parameters as variables.

:initializer
: *Value* is again either a form or the name of a function. If an initializer is provided, it will be called on each object as it is about to be allocated, whether the object was just created or is being reused. If *value* is a function, its arguments will be the resource data structure, the allocated object, and the supplied parameters. If *value* is a form, it may reference the parameters as variables, and also the allocated object, via the variable **object**.

:checker

A form or the name of a function. The job of the checker is to determine whether a given existing object is safe to allocate. If no checker is specified, the default action is to consider an object safe if it is not currently in use (*i.e.*, has not been allocated without being deallocated). If you specify a checker it will be used instead. A function here will be passed arguments of the resource data structure, the existing object being considered for allocation, the value of **in-use-p** for that object, and the supplied parameters. A form may reference the parameters as variables; the object under consideration, as **object**; and **in-use-p**. As you can see, the *free list* for a resource is a somewhat hypothetical object. When you ask to allocate an object, all of the existing objects are initially eligible. The default checker creates the functional equivalent of a free list by approving only those objects which are in fact free, but you needn't have this behavior. Supplying your own checker will change it. If, for instance, your checker always returned `t`, a given object could be simultaneously in use in any number of places, because it would always be considered safe for allocation, regardless of whether the previous allocater had deallocated it. And if your checker always returned nil, no object would be reused; every allocation request would result in the construction of a new object.

:matcher

A form or the name of a function. If no matcher is specified, an object is considered to satisfy the supplied parameters if they are **equal** to the parameters supplied at the time the object was constructed. If you specify a matcher, it will be used instead. A function here will be passed arguments of the resource data structure, the existing object being considered for allocation, and the supplied parameters. A form may reference the parameters as variables, and the object under consideration, as **object**.

:finder

A form or the name of a function. If this option is provided, the usual method for finding an object to allocate will not be used. The finder will instead be expected to somehow come up with an object. The checker, matcher, and constructor will not be called, unless the finder does so explicitly. A form or function specified here will see the same arguments as the constructor would.

:initial-copies

Value is a number, defaulting to 0. The specified number of objects will be constructed when the **defresource** is evaluated, thus creating an initial free pool of unallocated objects. If a resource has parameters and one or more initial copies are specified, the parameters must all be optional; the initial copies will have the default values of the parameters.

:free-list-size

Value is a number, defaulting to 20. It specifies the size of the array used to contain all the objects. (If the number of objects ever exceeds this size, the array is automatically replaced with a larger one.) **:free-list-size** is a misnomer, since the array contains both the free objects and the ones that are in use. As the earlier discussion of the `:checker` option pointed out, there isn't really any free list at all.

For all of the options which accept a form or the name of a function, if a form is supplied it is compiled during the evaluation of **defresource**, and put on the property list of the name of the resource.

allocate-resource *resource-name* &rest *parameters*

An object is allocated from the specified resource, matching the given parameters. The exact procedure followed depends on which options were supplied to **defresource** for this resource. If there is a finder, it is called and whatever it returns is used. Otherwise the set of existing objects is searched for one which satisfies both the checker (by default, is not in use), and the matcher (by default, was constructed with parameters **equal** to the current ones). If none are found, the constructor is called to create one. Finally, no matter which of the three routes yields an object, the initializer (if any) is called on it, and the object is marked as being in use.

deallocate-resource *resource-name object*

The object is conceptually returned to the specified resource's free-list, *i.e.*, its in-use marker is turned off.

using-resource (*variable resource parameters ...*) *body ...*

This macro, which calls **allocate-resource** and **deallocate-resource**, is preferred over calling those two functions directly. The *body* forms are evaluated inside a context where *variable* is bound to an object allocated from *resource* with the specified *parameters*. The object is deallocated at the end. An **unwind-protect** is used to guarantee that the object is deallocated before **using-resource** is exited.

Now an example. We define a resource of 2D arrays, with parameters for the number of rows and columns, which default to 100 each. A matcher is provided which accepts any array whose dimensions are at least as great as the given parameters. (The default matcher would require that the dimensions be exactly the same, meaning that we would very rarely reuse an object.) And an initializer fills the array with 0's.

```
(defresource sloppy-2D-array (&optional (rows 100) (columns 100))
  :constructor (make-array (list rows columns))
  :matcher (and (≥ (array-dimension-n 1 object) rows)
                (≥ (array-dimension-n 2 object) columns))
  :initializer (fillarray object '(0)))
```

And to use our resource:

```
(defun do-complex-computation (x y)
  (using-resource (temp-array sloppy-2D-array x y)
    ...
    (setf (aref temp-array i j) (calculate-value i j))
    ...))
```

There are several other built-in functions for dealing with resources.

deallocate-whole-resource *resource-name*
Deallocates all allocated objects of the specified resource. Use with caution, as it can lead to allocation of objects which somebody else is still using.

clear-resource *resource-name*
Makes the resource forget about all its existing objects. Future calls to **allocate-resource** will result in creation of new objects. Useful if the resource has been changed so that the old objects are no longer usable, or if some of the old objects have been damaged.

map-resource *resource-name function* &rest *args*
Applies *function* to each object known about by the resource. The arguments to *function* will be: the object, the value of **in-use-p** for the object, *resource-name*, and any additional arguments as specified by *args*.

8.2 Systems

The *system* facility provides a mechanism for keeping track of multiple files which together make up a single program. The group of files taken together is defined to be a system with the **defsystem** macro. Loading and/or compiling some or all of the files in a system is accomplished via the **make-system** function.

The system facility is far more complicated than the resource facility, and not necessarily very well-designed ("hairier than a breadbox," in the local parlance). It is supposed to be completely redone in a forthcoming Symbolics software release, so it's probably not worth your trouble to learn the more difficult features just now. We'll talk about some of the basic features; just enough for you to be able to use systems for your own programs.

The documentation on systems is Part II of volume 4, and is also quite extensive but not necessarily very well-designed, so this portion of the notes will follow the manual less closely than the resource section did.

Here is a typical call to **defsystem**:

```
(defsystem bar
  (:pathname-default "q:>george>")
  (:module reader-macros "rdmac")
  (:module other-macros "macro")
  (:module main-program "main" "commands")
  (:compile-load reader-macros)
  (:compile-load other-macros (:fasload reader-macros))
  (:compile-load main-program
                 (:fasload reader-macros other-macros)))
```

All four files involved are in the "george" directory on host "q." They are divided into three *modules*: reader-macros and other-macros, which consist of one file each, and main-program, which contains two files. The reason for this particular division is that it reflects the dependencies among the files, as specified in the `:compile-load` clauses. Each `:compile-load` clause states that the files in the specified module should be compiled if necessary (if the newest source file is more recent than the newest object file), and then loaded if necessary (if the newest object file is more recent than the last one to have been loaded), possibly subject to certain dependencies.

The reader-macros module (file "rdmac") does not depend on any other modules. The other-macros module (file "macro"), on the other hand, does depend on reader-macros. The `(:fasload reader-macros)` dependency for other-macros means that the file(s) in reader-macros have to be loaded before those in other-macros may be compiled or loaded. The reason is presumably that the file(s) in other-macros contain calls to macros defined in reader-macros, which must be loaded for the calling functions in other-macros to compile correctly. The files in main-program are dependent on both the other two modules. Presumably they contain calls to macros defined in both reader-macros and other-macros, and so require that both modules be loaded before main-program may be compiled.

Once the system "bar" has been defined in this manner, it can be loaded and/or compiled with **make-system**. `(make-system 'bar)` will load any files that need to be loaded, without doing any compilation. `(make-system 'bar :compile)` will first do any compilations that are needed, and then do any loading that is necessary. By default, **make-system** asks for confirmation before actually doing any compiling or loading.

One bit of terminology: the operation of compiling or loading an individual file is called a *transformation*. So a **defsystem** could be seen as defining what transformations a system is composed of, and a **make-system** as a command to see which of the transformations are necessary, and to carry them out.

Now we'll go into some more of the various options for **defsystem** and **make-system**. Many will be skipped entirely.

For **defsystem**:

We've already seen **:pathname-default**, **:module**, and **:compile-load**.

:component-systems

The mechanism for including other defined systems as parts of this one. What follows the keyword is a list of systems. When a **make-system** is done on this system, it will also be done on each of the component systems. By default, the transformations for this system will be performed before the transformations for each of the component systems. (Yes, that seems wrong to me, too.) But the default ordering can be overridden. If some of the local transformations depend on having the component systems done first, you can use `(:do-components nil)`. Put it at any position in the body of the **defsystem**, and the transformations of the component systems will be performed at a time corresponding to the chosen position in relation to the local system's `:compile-load` transformations.

:package

Specifies a package in which the transformations are to be performed, overriding any package specifications in the attribute lists of the individual files.

:patchable

Allows the system to be patched* (incrementally updated). Appropriate for

* PATCH

1. *noun.* A temporary addition to a piece of code, usually as a quick-and-dirty remedy to an existing BUG or MISFEATURE. A patch may or may not work, and may or may not eventually be incorporated permanently into the program.

large systems which are to be distributed to many users. The latest patches for a patchable system may be loaded with the **load-patches** function.

And for **make-system**:

(Recall that the **:compile** keyword must be specified for any compilations to occur. By default, only loading is done.)

:noconfirm
Assumes a yes answer for the questions **make-system** would otherwise ask before performing transformations.

:print-only
Performs no transformations at all; just prints information about which transformations would need to be done.

:version
Loads a particular version of a patchable system. There are many different ways to specify which version — see the documentation.

One last issue remains with respect to systems. Since **make-system** only works after the corresponding **defsystem** has been evaluated, it's important to have a convenient way to get the **defsystem** done. Knowing what file it's in and loading that file manually before doing a **make-system** for the rest is not convenient. Fortunately, there is something better. Whenever **make-system** is called on an unknown system, *i.e.*, one for which a **defsystem** has not yet been done, **make-system** looks in a predetermined place for a file to help it out. If there is a file named "sys: site; *system-name*.system" (a logical pathname whose physical translation depends on your site), **make-system** will first load that file, and then try to make the system.

The file should contain either the **defsystem** itself, or a call to **si:set-system-source-file**, which will tell **make-system** what file does contain the **defsystem**. The two arguments to **si:set-system-source-file** are the name of the system and the file where the system definition may be found. If you're the only person likely to be using the system, another idea would be to put the call to **si:set-system-source-file** in your login init file, thus eliminating the need to put a special file in the "sys: site;"

2. *verb.* To fix something temporarily; to insert a patch into a piece of code. See KLUGE AROUND.

(*The Hacker's Dictionary*, Guy L. Steele, Jr., *et al*)

directory.

8.3 Problem Set #8

Questions

1. Find out about **defwindow-resource**. It's used many times in the file "sys: window; sysmen", which contains the code for the system menu. It's also used in the "Smiling Face" example, "sys: examples; smile". (If you have Release 5 documentation, you can find a description of this example, including a few comments on **defwindow-resource**, in WINDEX). Now think of some sort of window for which it would be useful to have a resource, define it with **defwindow-resource**, and use it.

2. Look through the "sys: site;" directory. Pay close attention to the system definition files and the logical host definition files. (Recall that system definitions are by default found in "sys: site; *sys-name*.system" and logical host definitions in "sys: site; *host-name*.translations". Note also that if the "sys:" logical host happens to translate at your site to a UNIX system with 14 character limit on filenames, then "*sys-name*.system" will be translated to "*sys-name*.sy" and "*host-name*.translations" to "*host-name*.ld".)

   Most of the system def files will probably not directly include the **defsystem**, but will have a call to **si:set-system-source-file**, specifying some other file as the system source file. Track down a few of these pointers, and examine the defsystems. (Keep in mind that if **si:set-system-source-file** uses a logical pathname, the logical host must be defined with **fs:make-logical-pathname-host** before you can find the file.) Assure yourself that you understand how it is that **make-system** can find the defsystem for a new system. (You might want to look at the source code and see how **make-system** calls **find-system-named**, which calls **maybe-reload-system-declaration**, which calls **make-system-site-file-pathname**.)

Solutions (roughly speaking)

1. There isn't much to say about this.

2. Let's consider what happens if you do `(make-system 'ip-tcp)` on a machine that so far knows nothing about "ip-tcp." Since a system named "ip-tcp" has not been defined, **make-system** will look in the "sys: site;" directory for a file named "ip-tcp.system". At my site, the logical pathname "sys: site; ip-tcp.system" translates to the physical pathname "l:>sys-6>site>ip-tcp.system". Having found that file, make-system will load it. Here's what's in the file:

```
;;; -*- Mode: LISP; Package: USER; Base: 10 -*-

(fs:make-logical-pathname-host "IP-TCP")
(si:set-system-source-file "IP-TCP" "IP-TCP: IP-TCP; SYSTEM")
```

   The second form tells make-system where to find the **defsystem** for ip-tcp. But the file is specified as a logical pathname, using host "ip-tcp," which is not yet defined. Thus the necessity of the first form, the **fs:make-logical-pathname-host**. Recall that this function, when given an unknown host, also has a specific place to look for a file to define the logical host: "sys: site; ip-tcp.translations". At my site again, that translates to "l:>sys-6>site>ip-tcp.translations". So as part of the evaluation of (fs:make-logical-pathname-host "ip-tcp"), that file will be (recursively) loaded. And here's what's in it:

```
;;; -*- Package: USER; Base: 10; Mode: LISP -*-

(fs:set-logical-pathname-host
  "IP-TCP"
  :physical-host "L"
  :translations '(("IP-TCP;" ">sys-6>ip-tcp>")
                  ("IP-TCP;PATCH;" ">sys-6>ip-tcp>patches>")))
```

   Now things begin to bottom out. This form defines the logical host "ip-tcp," and specifies which directories on physical host "l" (Laphroaig, one of our lisp machines) the logical pathnames should translate to. As the loading of this file is completed, the call to fs:make-logical-pathname-host will also return, and we continue on to the **si:set-system-source-file**. But now the logical pathname "ip-tcp: ip-tcp; system" is meaningful, and translates to "l:>sys-6>ip-tcp>system". So make-system is now informed as to which file contains the defsystem for ip-tcp, and proceeds to load it. Here's what's in that file:

```
;; -*- Mode: Lisp; Package: TCP; Base: 10 -*-
;; DOD Internet Protocol support.
```

 ... *Symbolics' copyright notice* ...

```
(defsystem ip-tcp
  (:name "IP-TCP")
  (:maintaining-sites :scrc)
  (:pathname-default "IP-TCP: IP-TCP;")
  (:patchable "IP-TCP: IP-TCP; PATCH;")
  (:component-systems
             tcp tcp-service-paths ip-tcp-applications)
  (:module chaos "chaos-unc-interface")
  (:module global "ip-global")
  (:module main ("ip" "ip-routing"))
  (:module ip-protocols ("icmp" "udp" "egp"))
  (:compile-load chaos)
  (:compile-load global)
  (:compile-load main (:fasload global))
  (:compile-load ip-protocols (:fasload global))
  (:do-components (:fasload global)))
```

 ... *defsystems for tcp, tcp-service-paths, and ip-tcp-applications* ...

There's all kinds of stuff at the tail end of this file, which is a perfectly legitimate short-cut. We can assume that it will all be read the first time someone does a `(make-system 'ip-tcp)`. But the crucial part is that this file must contain the defsystem for ip-tcp, which you can see above. Once the loading of the file is complete, make-system will continue. It now knows how the ip-tcp system is defined, and can proceed to load the files which compose the system. The filenames are all specified with ip-tcp logical pathnames, but that's okay since we've already defined the ip-tcp logical host.

This sort of bootstrapping may seem awfully baroque, but notice that the whole mess can be transferred to some other site with a minimum of effort. The only line of code that would need to be changed is `:physical-host "L"` in the "sys: site; ip-tcp.translations" file.

Chapter 9

SIGNALING AND HANDLING CONDITIONS

9.1 Overview

The material for this chapter comes entirely from Part XI of volume 2 of the Symbolics documentation.

An *event* is some set of circumstances a running program can detect. Some events are classified as errors, but not all are. Division by zero is an event which is an error. When an event occurs, the program reports that fact, and finds some user-supplied code to execute as a result.

The reporting process is called *signaling*, and the user-supplied code which subsequently assumes control is a *handler*. The system software includes default handlers for a standard set of events.

The mechanism for reporting an event relies on flavors. Each class of events corresponds to a flavor, called a *condition*. Signaling is more fully described as *signaling a condition*, which involves creating a *condition object* (an instance of the appropriate flavor). When division by zero occurs and is signalled, the condition object created is an instance of the flavor **sys:divide-by-zero**. The instance variables of the condition object will contain information about the event, including what string to use if an error message becomes necessary.

Each handler is defined to be applicable only for a particular flavor of condition object. It can be used only when the condition signalled is an instance of that flavor, or one built on it. The set of conditions a handler can handle is thus determined by the flavor inheritance mechanism. Handlers have dynamic scope, so finding a handler to invoke for a given condition involves stepping through the stack and grabbing the first handler which is applicable to that condition.

There are several kinds of actions a handler can take. It may instruct the program to continue past the point where the condition was signalled, possibly after correcting the circumstances that led to the condition being signalled. This is called *proceeding*. Or it may unwind the stack all the way to the point where the handler was bound, flushing the pending operations. (This behavior is essentially equivalent to what you'd get with a **catch** in place of the handler, and a **throw** — with the correct tag — in place of the signaling of the condition.) Or it may partially unwind the stack to some intermediate point and reexecute from there. This kind of handler is called a *restart* handler.

There are three ways to customize the condition mechanism:

- Defining handlers for existing flavors of conditions which may be signalled by the system code.
- Signaling existing flavors of conditions within your code, which may invoke the system's default handlers or ones that you've written.
- Defining new flavors of conditions.

9.2 A Few Examples

```
(condition-case ()
    (// a b)
  (sys:divide-by-zero *infinity*))
```

This form binds a handler for the **sys:divide-by-zero** condition, and evaluates `(// a b)` in that context. If the division finishes normally, its value is returned from the condition-case. But if `b` turns out to be zero, the **sys:divide-by-zero** condition is signalled, and our handler is invoked, which simply causes the condition-case to return the value of the symbol `*infinity*`.

```
(defflavor block-wrong-color () (error))
```

```
(defmethod (block-wrong-color :report) (stream)
  (format stream "The block was of the wrong color."))
```

This defines a new error condition. (To define a condition which is not to be treated as an error, build it on the flavor **condition** rather than **error**.) The `:report` method is required. If your program now executes `(error 'block-wrong-color)`, your new condition will be signalled. If there are no handlers currently bound for this condition, a default handler will cause an entry into the debugger and use the `:report` method to generate the error message. This particular error message, however, is not terribly informative. It would be nice to know which block had the wrong color, and what color it had. Here:

```
(defflavor block-wrong-color (block color) (error)
  :initable-instance-variables
  :gettable-instance-variables)

(defmethod (block-wrong-color :report) (stream)
  (format stream
          "The block ~S was ~S, which is the wrong color."
          block color))
```

Now that we've added the `block` and `color` instance variables, we can do something like

```
(error 'block-wrong-color :block the-bad-block
                          :color the-bad-color)
```

which will initialize the instance variable as specified, and thus allow the `:report` method to get at that information.

Another function which may be used to signal conditions is **signal**. It has the same syntax as **error**, but may be used with any flavor of condition, whereas **error** is restricted to use with error conditions, *i.e.*, conditions built on the flavor **error** (which is indirectly built on the base flavor **condition**, via the intermediate flavor **dbg:debugger-condition**). There are two additional differences. **signal** allows handlers to *proceed* the condition, and **error** does not. (Thus **error** is guaranteed never to return to its caller.) And when called on a *simple condition* (*i.e.*, one not built on **dbg:debugger-condition**, as the flavor **error** is), **signal** returns nil if there are no handlers currently bound for the condition. (This is actually more a difference in the behavior of different flavors of condition than a difference in the behavior of the functions **signal** and **error**, but since **error** can't be used with simple conditions, it works as well to think of it as related to the function used.) If you signal any condition built on **dbg:debugger-condition**, with either **signal** or **error**, and there are no handlers bound for that condition, you always end up in the debugger.

The function most frequently chosen for signaling is actually neither of these, but rather **ferror**. It's used to signal an unproceedable error when you don't particularly care which condition is utilized. (**ferror** eventually calls **error**, with a condition flavor of **ferror**.) It allows you to specify a format control string and arguments to be used to construct the error message.

The macros **check-arg** and **check-arg-type** are also very handy for signaling an error in case a function has been sent inappropriate arguments.

9.3 More on Handlers

The following is (I believe) a complete list of the macros provided for convenient binding of handlers: **ignore-errors, condition-case, condition-case-if, condition-bind, condition-bind-if, condition-bind-default, condition-bind-default-if, condition-call,** and **condition-call-if**. We will discuss four of the nine.

ignore-errors *body...*

This one's easy. It binds a handler which handles absolutely every kind of error condition (not simple conditions) and does absolutely nothing. Upon seeing the error, the **ignore-errors** form returns. There will be no indication that the error ever occurred (except, of course, that any code within the **ignore-errors** following the error will not have been executed). **ignore-errors** is intended to replace the older forms **errset** and **catch-error**.

condition-case (*vars...*) *form clauses...*

form is evaluated in a context with handlers bound as specified in *clauses*. If *form* returns without signaling any conditions, the **condition-case** also returns (subject to one exception — see below). If a condition is signalled, the *clauses* are checked for a handler bound for that condition. If one is found, the rest of that clause tells how to handle the condition. If a condition is signalled for which the **condition-case** has not bound any handlers, the signal continues up the stack.

Each clause is a list whose car is the name of a condition flavor (or list of condition flavors) and whose cdr is a list of forms to evaluate. If a condition is signalled matching the flavor(s) (*i.e.*, equal to it or built on it) specified in a clause, the "handler" consists of executing the forms *in the dynamic environment of the* **condition-case**, not the environment where the signal occurred. That is, the stack is automatically unwound before the handler is executed. As a result, the handler may not proceed the condition. While the handler is running, the first symbol in *vars* will be bound to the condition object.

As a special case, the car of the last clause may be `:no-error`. Then if no condition is signalled during execution of the body, instead of returning *form*'s return values, the *vars* will be bound to those values, the `:no-error` forms will be evaluated in that context, and **condition-case** will return whatever they return.

This example is essentially equivalent to `(ignore-errors (do-this))`:

```
(condition-case ()
    (do-this)
  (error nil))
```

And this one uses the condition object:

```
(condition-case (e)
    (time:parse string)
  (time:parse-error (format error-output
                            "~A, using default time instead."
                            e)
                    *default-time*))
```

condition-bind *bindings body...*

Condition-bind provides similar functionality to **condition-case**, with the additional capability of proceeding from conditions. Each binding in the list of *bindings* is a list of two elements, the name of a condition flavor (or list of flavors), and a form which produces a handler function. The form is typically a quoted symbol, with the symbol being given a function definition elsewhere. But the form may also be a lambda expression. The *body* consists of any number of forms. If a condition is signalled during the evaluation of the *body*, the bindings are searched just as with **condition-case**. If a match is found, the handler function is called with one argument, the condition object. The handler runs *in the dynamic environment in which the error occurred*; unlike with a **condition-case**, the stack is not unwound.

The handler function has three options. It may return `nil` to indicate that it doesn't wish to handle the condition after all. (The search will then continue for a willing handler.) It may use **throw** to unwind the stack to some outer **catch**. Or it may proceed the condition by returning a non-nil value. There are several things to keep in mind if you wish to proceed a condition. First, the condition must be of a type which is proceedable; second, the condition must have been signalled with **signal**, rather than with **error**; and third, the handler should send the condition the `:proceed` message (with an appropriate argument) and return whatever values the `:proceed` method returns, because the `:proceed` method may decline to actually proceed and return nil, in which case the handler should also return nil.

Apart from being able to proceed conditions, the other advantage of running in the environment where the condition was signalled is that you may examine the stack. A handler might choose from among its three options according to what it finds on the stack, or it might print some message whose contents is determined by the state* of the stack. Section 63.4 ("Application: Handlers Examining the Stack") discusses the functions which are available for this purpose.

condition-bind-default *bindings body...*

Beyond the regular bound handlers, you can also define *default* handlers, with **condition-bind-default**. The list of current default handlers is checked only after all the bound handlers have declined to handle a condition. Thus by setting up a default handler you can allow *outer* bound handlers to take precedence over your handler, but still have your handler invoked if there are no appropriate bound handlers. (See chapter 65, "Default Handlers and Complex Modularity.")

9.4 Restart Handlers

There are several macros for establishing *restart handlers*. Here's an example taken from the chaosnet code:

```
(defun connect (address contact-name
                &optional (window-size default-window-size)
                          (timeout (* 10. 60.))
```

* STATE *noun.*

Condition, situation. Examples: "What's the state of your latest hack?" "It's WINNING away." "The SYSTEM tried to read and write the disk simultaneously and got into a totally WEDGED state."

A standard question is "What's your state?" which means "What are you doing?" or "What are you about to do?" Typical answers might be "I'm about to GRONK OUT" or "I'm hungry."

Another standard question is "What's the state of the world?" meaning "What's new?" or "What's going on?"

(*The Hacker's Dictionary*, Guy L. Steele, Jr., *et al*)

```
                          &aux conn real-address (try 0))
  (error-restart
    (connection-error
      "Retry connection to ~A at ~S with longer timeout"
      address contact-name)
   forms...))
```

This function evaluates *forms* and returns the last value if successful. But if the debugger assumes control as a result of a **chaos:connection-error** condition during the evaluation, the user will be given the opportunity to restart by typing one of the super keys. The debugger printout will include a line that looks something like

```
s-A:  Retry connection to SCRC at FILE 1 with longer timeout
```

If the user then types s-A the body of the **error-restart** will be executed again from the beginning. Now the full descriptions of two of the macros that may be used to establish restart handlers:

error-restart (*condition-flavor format-string format-args...*) *body...*
: This form establishes a restart handler for *condition-flavor* and then evaluates the body. If the handler is not invoked, **error-restart** returns the values produced by the last form in the body and the restart handler disappears. When the restart handler is invoked, control is thrown back to the dynamic environment inside the **error-restart** form and execution of the body starts all over again. *condition-flavor* is either a condition or a list of conditions that can be handled. *format-string* and *format-args* are a control string and a list of arguments to be passed to **format** to construct a meaningful description of what would happen if the user were to invoke the handler. *format args* are evaluated when the handler is bound. The debugger uses these values to create a message explaining the intent of the restart handler.

error-restart-loop (*condition-flavor format-string format-args...*) *body...*
: Similar to **error-restart**, but with an infinite loop. If the handler is not invoked, instead of returning the body is reexecuted. (The loop may be exited with **return**.) This form is commonly used in the top-level function for a process, with *condition-flavor* being **(error sys:abort)**.

9.5 More on Proceeding

Chapter 68 ("Proceeding") is a cogent five-page discussion of what is involved in programming proceedable errors. I recommend reading it. I will include just a few highlights here.

For proceeding to work, two conceptual agents must agree:

- The programmer who wrote the program that signals the condition
- The programmer who wrote the **condition-bind** handler that decided to proceed from the condition, or else the user who told the debugger to proceed.

The signaller signals the condition and provides a set of alternative *proceed types*. The handler chooses from among the proceed types to make execution proceed. A proceed type is defined by giving the condition flavor a `:proceed` method. (`:proceed` methods are combined using the `:case` combination type, so that one flavor may have any number of `:proceed` methods, each defining a different type. The first argument to the method dictates which `:case` is to be called.)

The body of the `:proceed` method can do anything it wants, generally trying to repair the state of things so that execution can proceed past the point at which the condition was signalled. It may have side-effects on the environment, and it may return values (which will then be returned by **signal**) so that the function that called **signal** can try to fix things up. Its operation is invisible to the handler; the signaller is free to divide the work between the function that calls **signal** and the `:proceed` method as it sees fit.

Review: suppose a condition is signalled for which a handler has been bound with **condition-bind**. The handler function is called with one argument, the condition object, and it may throw to some tag, or return nil to decline to handle the condition, or try to proceed the condition. To proceed, it must first determine which proceed types are valid for the condition object. This must be done at run-time because condition objects can be created that do not handle all of the proceed types for their condition flavor, via the `:proceed-types` init option, and because condition objects created with **error** instead of **signal** will have no proceed types. The handler may use the `:proceed-types` message to get a list of the available proceed types, or it may use the `:proceed-type-p` message to check a particular candidate. Having chosen a proceed type, the handler sends the condition object a `:proceed` message with one or more arguments. The first argument is the proceed type, and the rest are the arguments for that proceed type. Sending the `:proceed` message should be the last thing the handler does. It should then return immediately, propagating the values from the `:proceed` method back to its caller. Determining the meaning of the returned values is the business of the signaller only; the handler should not look at or do anything with these values.

The **signal-proceed-case** macro provides a convenient way to signal a proceedable condition, choose which of the defined proceed types for that flavor of condition should be considered available to the handlers, and specify separate actions to take for each of the proceed types (after the `:proceed` method returns).

Chapter 10

THE MOVING ICONS EXAMPLE

The original version of the moving icons example was put together by Ken Church for a class he taught in 1984. I've made some cosmetic modifications, and updated it to take advantage of some more recently written support software. The basic idea is that you have a frame split into two panes: a command menu of icons the user may "pick up" with the mouse, and an initially empty pane where the user may drop icons that have been picked up.

As before, if your site has loaded the tape that comes with the book, do `Load System moving-icons` [or `(make-system 'moving-icons)`] to load the code. But this time, to make a frame and start the action, get the system menu and click on *Moving Icons* at the lower right. Click left over one of the icons to pick it up, move the mouse over the main pane, and drop the icon by clicking left again.

10.1 General

It's my impression that there are four concepts in this short piece of code which are new: using Zmacs-style *command tables* ("comtabs"), pop-up mini-buffers, *typeout windows*, and changing the appearance of the mouse blinker. Most of the dirty work involved in getting the first three to work is taken care of by the "comtab" system (locally written software which both Ken and I have worked on). The **zwei:window-with-comtab** flavor captures most of the fruits of this work, and is itself quite easy to use. Another portion of the support code redefines many internal editor functions for reading typein from the mini-buffer to also work outside the editor, with pop-up mini-buffers.

A comtab associates characters with functions, so that whenever a certain key is pressed the corresponding function is called. In the editor, for instance, #\c-F is bound to the function **com-forward**, which moves the cursor forward. ("Characters" includes mouse characters, so functions may also be bound to particular mouse clicks.) In addition to single key commands, a comtab may include *extended commands*. These are accessible via Meta-X. You then type in the full name of a command to a mini-buffer. (Completion is active, so you frequently need only to type a few characters.) If you're in the editor proper, the permanent mini-buffer at the bottom of the screen is used. If you're using the **window-with-comtab**, a temporary mini-buffer pops up.

The **window-with-comtab** flavor has a `comtab` instance variable, and also includes the flavor **tv:process-mixin**. The top-level function for the process is a loop which reads characters and looks them up in the window's own comtab, calling the functions that are found. (It also does something appropriate if it sees a blip instead of a character.) All you need do is define the functions and put them in the comtab. By copying so closely the mechanism used by the editor, we are able to use many of the editor functions without change (or with small changes), most notably the on-line documentation features. The Help key is active in any window built on window-with-comtab, providing various functions for finding out about the currently defined commands. And a number of extended commands have also been brought over from the editor, including Edit Key, Edit Extended Command, Lookup Key Bindings, Describe Key Bindings, Set Variable, and List Variables.

Any window built on **window-with-comtab** will also have a *typeout window* associated with it. Just as in the editor, if you hit the Suspend key, the typeout window will become exposed, and you will enter a break loop, so that you may type arbitrary lisp forms to be evaluated.

Let's take a look at the code now.

10.2 moving-icons-frame

We have two panes, the command menu and the main pane. The frame is a bordered constraint frame with shared io-buffer, so that the process running in the main pane will see the blips from the command menu. The only new stuff is in the default-init-plist specified for the menu. The most interesting is the specified value for `:item-list`. As in the tree example of three weeks ago, we've used the "general" item type. Each element of the list, corresponding to one item in the menu, is a list of five elements. The first is the string which will be printed to display the item in the menu. As you can see from the form of the loop, the string will be one character long for every item, with the character code ranging from 0 to 127. Since each item is of type `:eval`, the effect of executing an item will be to evaluate the form following the `:eval` keyword. And the form sends the main pane the `:pick-up-icon` message, with an argument of the item's character code. (It helps to know that the **window-with-comtab** code takes care of binding the variable **zwei:*window-with-comtab*** to the object which has the **window-with-comtab** mixin, in this case the main pane.) So if I choose the item in the upper left corner of the menu, the main pane will receive a `:pick-up-icon` message with an argument of 0.

The `:rows` init spec instructs the menu to display its 128 items in 4 rows of 32 each, implying that it must do some horizontal spreading. And the `:default-font` spec is crucial. Since the items are really just one-character strings, the way they appear depends on what font the menu uses for printing them. By default, command menus use a standard variable-width alphabetic font called `JESS14`. But we've specified the font named `MOUSE`, which happens to contain many of the characters which are used for the mouse blinker in various situations. You could use any other font — either an existing one or one of your own creation — by changing the `:default-font` spec (or by sending the `:set-icon-font` message, which is done by Meta-X Set Icon Font). You can read about using fonts in section 12.7 of volume 7. See also the Zmacs commands Meta-X List Fonts and Meta-X Display Font. For modifying a font or creating a new one, use the Font Editor, which is described in Part II of volume 3.

10.3 moving-icons-main-pane

This flavor is built on **zwei:window-with-comtab**, with **tv:pane-no-mouse-select-mixin** added, to make it a pane, and one which will not confuse the select menu. An instance variable keeps track of the current icon. The default-init-plist specifies what comtab to use (***icon-comtab*** is defined at the end of the file) and what font the window should print with. This font obviously needs to match the font of the command menu.

The first two methods are quite simple. The `:set-icon-font` method sets the default font of the command menu (which causes it to redisplay all the items in the new font) and sets the window's own font-map to match (which affects only subsequent typeout — the current display is left alone). The `:pick-up-icon` message, recall, is sent when a menu item is executed. The method sets the icon-character instance variable, and sends the window the `:mouse-standard-blinker` message, which we'll see has the effect of making the mouse blinker look like the item which was just chosen.

10.4 Messing with the mouse blinker

The easiest way to alter the appearance of the mouse blinker (though not the way we do it in this example) is with the function **tv:mouse-set-blinker**. It has one required argument — a keyword symbol — and two optional ones. The keyword tells what sort of blinker to switch to, and the system must already have been told how to get a blinker of that type. Teaching the system about new types of blinkers is not difficult, but it's also not necessary in this case, so we don't need to go into it. The reason it's not necessary is that there's already a kind of blinker which is sufficiently flexible for our needs, and that is the `:character` blinker.

This sort of blinker has two instance variables, one specifying a font to use and one a character code. The blinker draws itself as the given character in the given font. So by adjusting the values of the instance variables, a single blinker can be made to look like any character in any defined font. This is exactly the reason for the existence of the "mouse" font; most of the various mouse blinkers you're used to seeing are actually the same `:character` blinker, set to varying characters in the mouse font.

And finally we come to the function **tv:mouse-set-blinker-definition**, which is what the example uses in place of **tv:mouse-set-blinker**. The former takes several additional arguments, which allow you to send an arbitrary message to the "new" blinker (which may really be the same object as the old blinker). The message we send is `:set-character`, which takes two arguments, the new character code, and (optionally) the font to use. In our case the character is the value of the instance variable icon-character, which will have been set by the `:pick-up-icon` method, and the font is the main pane's current font (accessed via the instance variable `tv:current-font`), which we have been careful to make sure is always the same as the menu's font. For a slightly different effect, type this at a lisp listener:

```
(tv:mouse-set-blinker-definition :character 0 0 t :set-character
#\M 'fonts:43vxms)
```

The second and third arguments to **tv:mouse-set-blinker-definition** specify the *x-offset* and *y-offset* for the blinker. By default, the offsets are 0, meaning that the blinker draws itself with its upper left corner at the actual mouse position. Often you would rather the blinker position itself differently, perhaps with its center over the real mouse position. That's when you need to set non-zero offsets. In the example, we use whatever offsets the previous blinker had, to minimize any apparent motion of the mouse as we switch blinkers. And if you glance down at the `:drop-icon` method you'll see we need to adjust for the offsets again when drawing the icon on the main pane, so that the icon is placed exactly under where the mouse *appears* to be.

But let's return to the `:mouse-standard-blinker` method. Why did I make the `:pick-up-icon` method call this one to change the mouse blinker, instead of just including the multiple-value-bind form directly in `:pick-up-icon`? And why does `:mouse-standard-blinker` test whether `icon-character` has a non-nil value when `:pick-up-icon` will have just given it one? The answer to both questions is that the mouse process will occasionally be sending our window the `:mouse-standard-blinker` message, and I want to do something appropriate then as well as when `:pick-up-icon` sends the message. Exactly when the mouse process sends `:mouse-standard-blinker` is difficult to explain, but a reasonable approximation is that it will happen whenever the mouse crosses *into* the window. There is a default handler for `:mouse-standard-blinker`, which our window would have inherited, and its action is simply to pass the same message up to the window's superior. (If all of the window's superiors also follow the default route of passing the message up, it will eventually reach `tv:main-screen`, which will restore the blinker to the familiar NNW arrow seen in a lisp listener.) But our method supersedes the default one. If there is no current icon-character, ours behaves just like the default; if there is an icon-character, ours makes the mouse blinker look like the icon. Had I put the **tv:mouse-set-blinker-definition** in `:pick-up-icon` and kept the default method for `:mouse-standard-blinker`, we would have lost the special appearance of the mouse blinker every time it crossed into the main pane, including the mandatory first trip down from the command menu.

10.5 The :drop-icon method

Dropping the icon involves setting the cursor position to the correct spot, outputting the character, resetting the icon-character instance variable, and restoring the mouse blinker to its usual form. The only tricky step is setting the cursor position — the mouse position will be provided in terms of outside coordinates. These must be converted to inside coordinates by subtracting the margin widths, and then corrected for the blinker offsets.

10.6 Setting up the comtab

The way to define a command is with the **defcom** macro (in the zwei package). The first argument is the name of the command, which should begin with "com-". Then there's a documentation string, which the help key will know how to get at. The third argument is a list of options which are not relevant outside the editor. The rest is the body of the command. Note that there is no "argument list"; functions defined with **defcom** are intended to look for their arguments as the value of **zwei:*numeric-arg***. Recall that "arguments" to editor functions are numbers, struck with the control or meta keys down before the main command character is pressed. The effect of the numeric control keys is to bind ***numeric-arg*** appropriately.

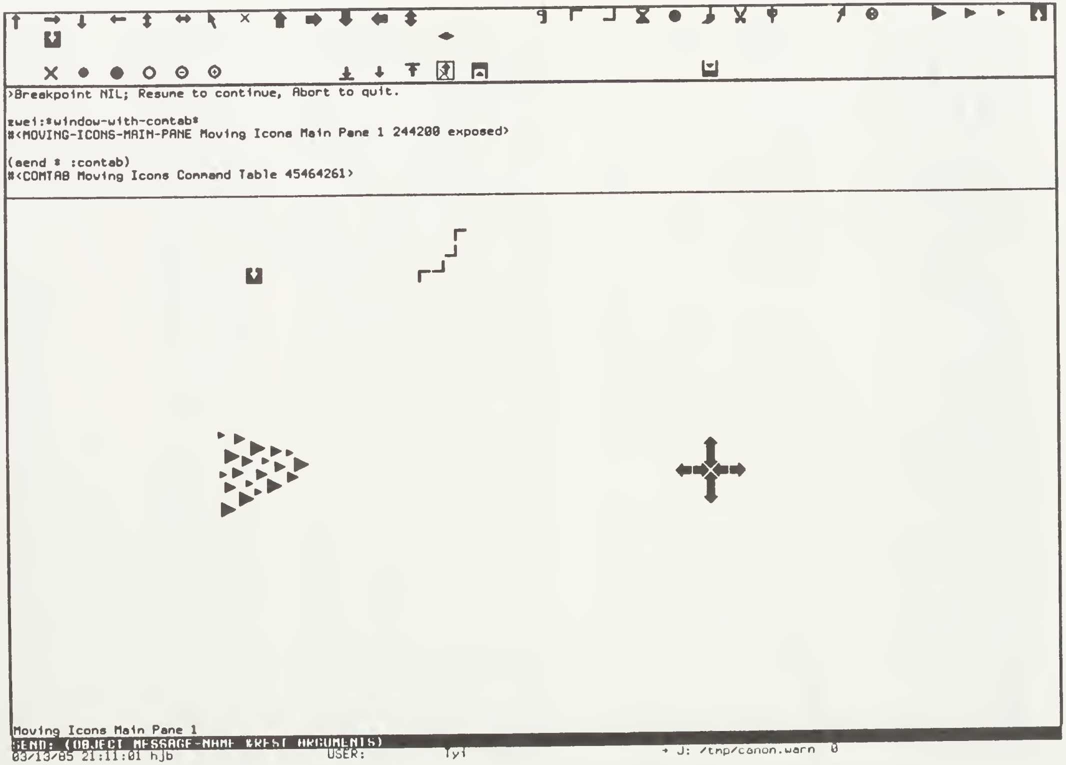
Executing a **defcom** just defines a function and adds the name of the command and its documentation to a list of all commands, but does not make the command usable — it is not included in any comtab. What puts the command into a comtab is the **zwei:set-comtab** function. **set-comtab** adds a list of character-command pairs (its second argument) to an existing comtab (the first arg). The third argument, if present, is a list of extended commands, *i.e.*, commands that will be accessible via Meta-X. The list should be created with **zwei:make-command-alist** and may be appended with the extended command list of some other comtab. A single command may appear in a comtab any number of times, paired with different characters and/or as an extended command. And it may simultaneously appear in any number of different comtabs.

The function **zwei:set-comtab-indirection** may be used to cause inheritance from another comtab. Any characters not found in the current comtab will be looked up in the second one. In particular, we need to inherit from ***basic-comtab*** (the one defined with **window-with-comtab**) the numeric argument commands and the help functions.

10.7 Getting in the System Menu

The last form in the file is what puts "Moving Icons" in the rightmost column of the system menu. The function **tv:add-to-system-menu-programs-column** takes three arguments. The first, a string, is what you want to appear in the menu. The second is a form to be evaluated should your item be chosen from the menu. This is usually a call to the function **tv:select-or-create-window-of-flavor**, with the one argument being the flavor of window you want created. Finally there's a documentation string, which appears in the mouse documentation line whenever the mouse is over your item in the system menu.

10.8 The Program



```
>Breakpoint NIL; Resume to continue, Abort to quit.

zwei:*window-with-comtab*
#<MOVING-ICONS-MAIN-PANE Moving Icons Main Pane 1 244200 exposed>

(send * :comtab)
#<COMTAB Moving Icons Command Table 45464261>
```

```
Moving Icons Main Pane 1
SEND: (OBJECT MESSAGE-NAME &REST ARGUMENTS)
03/13/85 21:11:01 hjb          USER:      Tyi           + J: /tmp/canon.warn  0
```

```
;;; -*- Syntax: Zetalisp; Mode: LISP; Package: (icons global); Base: 10 -*-


(defflavor moving-icons-frame
        nil
        (tv:bordered-constraint-frame-with-shared-io-buffer)
  (:default-init-plist
     :selected-pane 'main
     :panes `((menu tv:command-menu-pane
                    :rows 4
                    :default-font fonts:mouse
                    :item-list ,(loop for i below 128
                                      collect (list (string i)
                                                    :eval `(send zwei:*window-with-comtab*
                                                                 :pick-up-icon ,i)
                                                    :documentation "Pick up this icon.")))
              (main moving-icons-main-pane))
     :configurations '((main-configuration
                          (:layout (main-configuration :column menu main))
                          (:sizes (main-configuration (menu :ask :pane-size)
                                                      :then (main :even)))))))

(defmethod (moving-icons-frame :name-for-selection) nil
  tv:name)

(defflavor moving-icons-main-pane
        ((icon-character nil))                  ; ascii code of current icon (nil if no icon)
        (tv:pane-no-mouse-select-mixin
         zwei:window-with-comtab)
  :settable-instance-variables                  ; (implies gettable and initable)
  (:default-init-plist :comtab *icon-comtab*
                       :font-map '(fonts:mouse)
                       :blinker-p nil
                       :save-bits t))
```

```
(defmethod (moving-icons-main-pane :set-icon-font) (new-font)
  (send (send tv:superior :get-pane 'menu) :set-default-font new-font)
  (send self :set-font-map (list new-font)))

(defmethod (moving-icons-main-pane :pick-up-icon) (icon)
  (setq icon-character icon)
  (send self :mouse-standard-blinker))

(defmethod (moving-icons-main-pane :mouse-standard-blinker) nil
  (if icon-character
      (multiple-value-bind (x-off y-off) (send tv:mouse-blinker :offsets)
        (tv:mouse-set-blinker-definition :character x-off y-off t
                                         :set-character icon-character tv:current-font))
      (send tv:superior :mouse-standard-blinker)))

(defmethod (moving-icons-main-pane :drop-icon) (mouse-x mouse-y)
  (if (not icon-character) (beep)
      (multiple-value-bind (x-off y-off) (send tv:mouse-blinker :offsets)
        (send self :set-cursorpos (- mouse-x x-off tv:left-margin-size)
                                  (- mouse-y y-off tv:top-margin-size)))
      (send self :tyo icon-character)
      (setq icon-character nil)
      (send self :mouse-standard-blinker)))


(zwei:defcom com-drop-icon "Drops the current icon, if any, at the mouse position" nil
  (send zwei:*window-with-comtab* :drop-icon zwei:*mouse-x* zwei:*mouse-y*))

(zwei:defcom com-set-icon-font "Changes the font of the icon menu" nil
  (send zwei:*window-with-comtab* :set-icon-font
        (cdr (zwei:COMPLETING-READ-FROM-MINI-BUFFER
               "New Font:"
               (LOOP FOR X BEING LOCAL-INTERNED-SYMBOLS IN "FONTS"
                     WHEN (AND X (BOUNDP X) (TYPEP (SYMEVAL X) 'FONT))
                     COLLECT (CONS (GET-PNAME X) (symeval X)))))))
```

```
(defvar *icon-comtab* (zwei:create-sparse-comtab "Moving Icons Command Table"))
(zwei:set-comtab-indirection *icon-comtab* zwei:*basic-comtab*)
(zwei:set-comtab *icon-comtab*
                 '(#\mouse-1-1 com-drop-icon)
                 (zwei:make-command-alist '(com-set-icon-font)))


(tv:add-to-system-menu-programs-column
  "Moving Icons"
  '(tv:select-or-create-window-of-flavor 'moving-icons-frame)
  "Select a window for drawing a picture with icons from a menu.")
```

10.9 Problem Set #9

Questions

1. Make it possible to individually erase icons that have been drawn on the main pane. (I think the best way to do so would be to use the mouse sensitive items facility. You might also try doing it on your own, without mouse sensitive items. That would probably entail duplicating some of the ms-item stuff, but the limited functionality required here shouldn't involve too much duplication.)
2. If you've chosen the mouse sensitive item route for (1), add some more functions to the menu. For example, one option might ask for a keystroke then change the icon to the one corresponding to that character code (in the same font). Another might prompt for the name of a font then redraw the icon in that font (with the same character code).
3. In an earlier version of the window-with-comtab support code, the typeout window didn't restore the image of the window underneath it upon being deexposed. The main pane therefore needed to be able to regenerate the picture. Though it's no longer strictly necessary to have such a `:redisplay` method, it's still a good exercise.* Your solution to (1) should give you some way of knowing what icons are currently displayed and where. Use this knowledge to write the `:redisplay` method — it should do a `:clear-window` and then redraw the current icons. While you're at it, make the `:redisplay` method accessible through the comtab, via `#\c-L` and `#\refresh`.

* Sorry for resorting to such a pedantic rationale. I didn't plan it this way, but at the last minute I ruined what had been a perfectly good problem through my own industrious improvements to the comtab code.

Hints

Don't forget to put your code in the "icons" package!

1. Adding mouse-sensitivity involves the following steps: mix **tv:basic-mouse-sensitive-items** into moving-icons-main-pane; make an item type alist with **tv:add-typeout-item-type**, and put it on main-pane's default plist; write the function(s) to be called when an item is chosen; and alter the `:drop-icon` method to use the `:item` or `:primitive-item` messages. (Processing the blip and calling the indicated function will be taken care of by the window-with-comtab mixin; your function will be called with one argument, the item.)

   To erase the icon, draw over it in xor mode. It's probably easiest to use the `:draw-char` message. Since the icons themselves are just character codes, and don't contain information about their screen position, you'll probably want your mouse-sensitive items to actually be some sort of data structure containing the x and y coordinates as well as the character itself.

2. Erase the icon as in (1), then redraw it with the new character or font. `:draw-char` is probably the easiest way again.

3. Assuming again that you've chosen the mouse-sensitive-items route, you'll need to get at the list of current items in order to redraw each of them. The list is the value of the instance variable *tv:item-list*. (Remember to grab the list before doing the `:clear-window`, because the `:clear-window` will set the list to nil.)

Solutions

1. The structure representing the item is a list of the character code, the current font, and the x and y coordinates. The advantage of including the font is that if Meta-X Set Icon Font is done, we can still properly erase old icons in the original font.

```
(defflavor moving-icons-main-pane
           ((icon-character nil))
           (tv:pane-no-mouse-select-mixin
            tv:basic-mouse-sensitive-items                      new
            zwei:window-with-comtab)
  :settable-instance-variables
  (:default-init-plist :comtab *icon-comtab*
                       :item-type-alist *icon-item-type-alist*
                       :font-map '(fonts:mouse)                ↑new
                       :blinker-p nil
                       :save-bits t))

(defvar *icon-item-type-alist* nil)

(tv:add-typeout-item-type *icon-item-type-alist* :icon "Erase"
                           erase-icon t "Erase this icon.")

(defmethod (moving-icons-main-pane :drop-icon)
           (mouse-x mouse-y)
  (if (not icon-character) (beep)
      (multiple-value-bind (x-off y-off)
          (send tv:mouse-blinker :offsets)
        (send self :draw-icon
              (list icon-character tv:current-font
                    (- mouse-x x-off tv:left-margin-size)
                    (- mouse-y y-off tv:top-margin-size))
              :add))
      (setq icon-character nil)
      (send self :mouse-standard-blinker)))

(defmethod (moving-icons-main-pane :draw-icon)
           (char-info item-action)
  (destructuring-bind (char font x y) char-info
    (send self :draw-char font char x y tv:alu-xor)
    (selectq item-action
         (:add (send self :primitive-item
                     :icon char-info x y
                     (+ x (send self :character-width char font))
                     (+ y (font-char-height font)
                          (send self :vsp))))
         (:delete (setq tv:item-list
                        (del #'(lambda (item list)
                                 (eq item (second list)))
                             char-info tv:item-list))))))

(defun erase-icon (char-info)
  (send zwei:*window-with-comtab*
        :draw-icon char-info :delete))
```

2.
```
(tv:add-typeout-item-type
  *icon-item-type-alist* :icon "Change Character"
  change-icon-char nil
  "Replace this icon with a different character.")

(defun change-icon-char (char-info)
  (let ((new-char (read-single-char)))
    (process-sleep 30)
```
When the typeout window is deexposed, it restores the window underneath to how it was when the typeout window was first exposed, so if we leave the typeout window exposed while we change the display, our changes will be lost when it does get deexposed.
```
    (send zwei:*typeout-window* :deexpose)
    (send zwei:*window-with-comtab*
          :draw-icon char-info :delete)
    (setf (first char-info) new-char)
    (send zwei:*window-with-comtab*
          :draw-icon char-info :add)))

(defun read-single-char nil
  zwei:(LET (CHAR)
         (TYPEIN-LINE "New char: ")
         (SETQ CHAR (TYPEIN-LINE-ACTIVATE
                      (EDITOR-INPUT :MOUSE :RETURN)))
         (TYPEIN-LINE-MORE "~:@C" CHAR)
         char))
```
this is borrowed from the code for Help-C (com-self-document)

```
(tv:add-typeout-item-type
  *icon-item-type-alist* :icon "Change Font" change-icon-font
  nil "Redraw this character in a different font.")

(defun change-icon-font (char-info)
  (let ((new-font (read-font-from-mini-buffer)))
    (send zwei:*window-with-comtab*
          :draw-icon char-info :delete)
    (setf (second char-info) new-font)
    (send zwei:*window-with-comtab*
          :draw-icon char-info :add)))
```

where `(read-font-from-mini-buffer)` is the body of **com-set-icon-font** starting at `(cdr ...)`, and com-set-icon-font should be changed to call read-font...

3. In addition to the redisplay, I've also thrown in a **com-clear-window** to remove all the current icons.

```
(defmethod (moving-icons-main-pane :redisplay) nil
  (let ((old-item-list tv:item-list))
    (send self :clear-window)
    (loop for (nil item) in old-item-list
          do (send zwei:*window-with-comtab*
                   :draw-icon item :add))))

(zwei:defcom com-redisplay
             "Regenerates display from current dropped icons"
             nil
  (send zwei:*window-with-comtab* :redisplay))

(zwei:defcom com-clear-window
             "Removes all dropped icons from the window" nil
  (send zwei:*window-with-comtab* :clear-window))

(zwei:set-comtab *icon-comtab*
                 '(#\c-L com-redisplay
                   #\refresh com-redisplay
                   #\clear-input com-clear-window))
```

Chapter 11

MORE ADVANCED USE OF THE EDITOR

The standard Zmacs commands are generally quite well documented by the on-line help facilities. So there should be no difficulty in becoming fluent in the use of the built-in commands simply by consulting the automatic documentation. Or, if you prefer, many of the more common built-in commands are described on paper, in Part I of volume 4.

The methods for adding new commands, on the other hand, are not documented so completely. It is upon that topic that this class will concentrate.

11.1 Keyboard Macros

keyboard macros allow you to bundle up any number of keystrokes and execute them all with one keystroke. (These actually are documented, but since they fit in with the rest of today's chapter, I thought we should look at them as well.) The Zmacs command "`c-X (`" starts a keyboard macro. Whatever keys you press from then up until you type a "`c-X )`" are remembered while they are executed. When you type the `c-X )` the macro will be defined. It can be re-executed by typing `c-X E`. The effect will be as though you had typed all the keystrokes in the macro definition (but faster). Giving a numeric argument to `c-X E` will cause the macro to be repeated that many times.

`c-X E` always executes the most recently defined macro, so if you define another macro with `c-X (`, the definition of the first one will be lost, unless you have previously saved it somehow. One way of saving a macro is to give it a name, with M-x Name Last Kbd Macro. Once a macro has been named, you can install it on a key with M-x Install Macro. The name of the macro and the keystroke on which to install it will be prompted for in the mini-buffer. From then on (until you deinstall the macro, or install some other command on the same key), typing that keystroke will execute the macro. When M-x Install Macro asks for the name of the macro to install, if you just type a carriage return instead of the name of a macro, the one most recently defined will be installed on the specified key. It's therefore unnecessary to ever name your macros, as long as you install them before defining another.

Here's a simple example, something which I often did while working on the early versions of these lectures, in the troff text formatter. I'll define a keyboard macro for inserting the troff directives for switching to an italic font and back, and install it on super-I.

`c-X (`	start keyboard macro definition
`\fI\fP`	insert the text
`c-B c-B c-B`	move the cursor back to the correct position for inserting italic text
`c-X )`	end macro definition
`M-x Install Macro`	[prompted with "Name of macro to install (CR for last macro defined):"]
`<return>`	[prompted with "Key to get it:"]
`s-I`	[menu pops up for choosing which comtab to use]
`<click on Zmacs>`	

The next step, one which I've been too lazy to ever take, but really should, would be to put something in my login init file to automatically define this keyboard macro every time I login. As things stand, I have to run through the above sequence of commands whenever I start writing a text file (unless the machine hasn't been booted since the last time I defined the macro). The way to define a keyboard macro in lisp code is with the function **zwei:define-keyboard-macro**. The first argument is the name the macro will have, the second (usually nil) indicates a repeat count, and the rest are the character codes for the keystrokes. Once the macro has been defined, you can insert it into a comtab. So if I weren't so lazy, I would add this to my init file:

```
(zwei:DEFINE-KEYBOARD-MACRO italic-font (nil)
  #\\ #\f #\I #\\ #\f #\P #\c-B #\c-B #\c-B)

(zwei:command-store (zwei:make-macro-command :italic-font)
                    #\s-I
                    zwei:*zmacs-comtab*)
```

If I were adding several macros at one time, and to the same comtab, I would use **set-comtab** rather than **command-store**. And as for the choice of comtab, specifying ***zmacs-comtab*** is equivalent to clicking on "Zmacs" in the menu that M-x Install Macro pops up, and ***standard-comtab*** is equivalent to clicking on "Zwei." Each editor has its own comtab, but they are all indirected to ***zmacs-comtab***, so putting a command there means that it will be accessible in all instances of the Zmacs editor, unless of course the command is shadowed by a binding to the same key in an individual editor's comtab. And ***zmacs-comtab*** is indirected to ***standard-comtab***, as are all the other zwei-based editors* (*e.g.*, Zmail and Converse). So a command inserted in ***standard-comtab*** will be available in all the zwei editors, unless shadowed.

11.2 Writing New Commands

Most extensions to the editor are not expressible as a sequence of keystrokes. For these you need to write a function, with **defcom**, and then add it to the comtab of your choice with **command-store** or **set-comtab**. Among the things you may want to do from your function are: insert text into a buffer, read text out of a buffer, get user input from the mini-buffer, and send text to the typeout window. All of these are reasonably straightforward, once you know about a few key variables and functions.

11.3 Buffers and Streams

The functions **zwei:open-editor-stream** and **zwei:with-editor-stream** open a bidirectional stream to an editor buffer. They are analogous to **open** and **with-open-file** in that **open-editor-stream** simply creates the stream and returns it, while **with-editor-stream** puts a call to **open-editor-stream** inside a useful wrapper, and so is preferable if your control structure allows it. (The wrapper in this case guarantees not a

* As I understand it, *eine* and *zwei*, apart from being "one" and "two" in German, were the names of the first two text editors written for lisp machines. They are acronyms, respectively, for Eine Is Not Emacs, and Zwei Was Eine Initially.

close, which isn't meaningful for editor streams, but a `:force-redisplay`, so any changes to the buffer will be apparent.) Either of these functions ultimately ends up calling **interval-stream**, which does a make-instance of flavor **interval-stream**. For more control over how the stream is made, you may wish to call **interval-stream** directly. But in general, **open-editor-stream** provides a good higher-level interface.

There is some documentation on these two functions in chapter 44 of volume 7, mainly on the various options for specifying which buffer the stream should point to, and where in the buffer it should initially point. You must specify at least one of the following options: `:interval`, `:buffer-name`, `:pathname`, `:window`, or `:start`. `:buffer-name` and `:pathname` are easy enough. If a buffer exists which matches the given information, it is used; if not, one is created (unless the `:create-p` option has been used to specify otherwise). Understanding the others requires knowing a bit more about the editor data structures.

The base flavor for all buffers is **zwei:interval**. The value of the variable **zwei:*interval*** will be the current buffer, an object whose flavor is likely to be something like **zwei:file-buffer**, which is indirectly built on **interval** (via **node**, **top-level-node** and **buffer**). Thus references to the current "interval" mean the current buffer. An interval may also be created to contain any portion of a buffer; some intervals are actually buffers, while many others are temporary objects used to point to arbitrary regions of text. An object of flavor **interval-stream** (see above) is simply a stream whose "peripheral device" is an interval.

The **interval** flavor has (among others) two instance variables, for the beginning and end of the text it refers to. These two are represented as *buffer pointers*, a type of object defined by the **zwei:bp** defstruct. A bp is a list consisting of three elements: an object of type *line* (defined by the **zwei:line** defstruct), an index into the line, and a keyword we needn't be concerned with here. A line, in turn, is a string (the text of the line) with all kinds of information in the string's array-leader, most importantly pointers to the next and previous lines in the buffer. So given an interval you can get bp's for the first and last character positions in the buffer, given a bp you can tell what line it refers to (as well as which character in the line), and given a line you can find the preceding and following lines.

The value of the variable **zwei:*window*** is an object of the type defined by the defstruct **zwei:window** (not to be confused with objects of type **tv:window**). It contains information about the portion of the buffer currently visible. Among its slots are a pointer to the interval (buffer) that window is displaying part of, a bp for the position of *point* (the cursor), a bp for the first character in the line currently displayed at the top of the screen, and a count of how many lines are visible. The window defstruct uses the `:array-leader` type, meaning that the object is an array, with all the slots going into the array-leader. The array itself contains a row of information for each line in the area the window maps to, including the "line" object.

The macro **zwei:point**, called with no arguments, returns the bp for the current point. As you might expect, the macro expands into `(ZWEI:WINDOW-POINT ZWEI:*WINDOW*)`, the accessor for the "point" slot of the current window. A similar macro, **zwei:mark**, returns a bp for the most recently dropped mark, which is another slot in the **window** defstruct.

Now back to the options for **open-editor-stream**. The ones we delayed discussing were `:interval`, `:window` and `:start`. We're now in a position to make sense out of these. If you use the `:interval` option, the value supplied should be an interval; you may use **zwei:*interval*** or any of a number of functions which exist to create an interval pointing to an arbitrary text area. The stream created will read from and write to the given interval. The `:window` option is similar; you provide a (zwei) window, and **open-editor-stream** returns a stream into that window. The `:start` option is a little more complicated. It may be used either alone or in combination with other options. If the value you provide is a bp, the stream will begin at the specified bp. In this case no other options need be supplied, and the stream's "end of file" will be at the end of the buffer containing the bp. (You can force some other termination point with the `:end` option.) The other possible values for `:start` are all keywords, and all require that some other option (like `:interval` or `:window`) indicate which buffer to use. The effect of the `:start` keyword will be to determine where within the buffer the stream will start. Among your choices are `:beginning`, `:end`, `:mark`, `:point`, and `:region`.

There are quite a few other options to **open-editor-stream** to control various details of its behavior, but the ones we've already seen are sufficient to write all sorts of useful applications. You'll find many examples in the editor code to use as models. These are likely to use **interval-stream** directly, rather than through **open-editor-stream**, because **open-editor-stream** is a relatively new feature. Your own functions will probably be clearer and easier to write if you use **open-editor-stream** (or **with-editor-stream**).

Here are a few trivial examples, to illustrate the basic concepts. (All assume the current package is `zwei`.)

```
(with-editor-stream (str :interval *interval*)
  (send str :string-out
            "surprise text inserted at end of current buffer"))

(with-editor-stream (str :interval *interval* :start :beginning)
  (send str :string-out
        "surprise text inserted at beginning of current buffer"))

(with-editor-stream (str :start (point))
  (send str :string-out "surprise text inserted at point"))

(with-editor-stream (str :buffer-name "mbox //usr//hjb// S:"
                         :start :beginning)
  (send str :line-in))       returns first line of buffer containing my mbox file
```

One slightly subtle point to keep in mind is that the variables ***interval*** and ***window*** are not global; they're bound partway down the stack in the Zmacs process. This makes no difference if you're writing editor commands, because they'll be executed in the same process. But it does mean you'll have to be careful if you write code intended to interact with the editor's buffers from another process, because those variables will then be unbound. In particular, of the four examples just given, the first three would bomb if evaluated in a lisp listener (the third because **point** has to access ***window***), while the fourth would work. All four work fine from the editor's typeout window (as long as the package is correct), since the typeout window's break loop is in the Zmacs process.

11.4 Reading from the Mini-buffer

Another set of tools often used in writing editor commands are the functions for reading from the mini-buffer. There are many — their names generally begin with "typein-line-," and the varieties differ in how they know when the typein is completed, and in what form they return the typein. Here are a few of them (all in the zwei package):

typein-line-readline *ctl-string* &rest *args*

> A prompt (created by **format** using *ctl-string* and *args*) is printed in the *typein line* (just above the mini-buffer — you've seen it used frequently), and keyboard input is directed to the mini-buffer. When the Return or End keys are pressed, a string is constructed out of all the preceding characters and returned. The behavior is analogous to that of the **readline** function.

typein-line-read *ctl-string* &rest *args*

Same as above except the lisp reader scans the string before it is returned, and the lisp object returned by the reader is returned from **typein-line-read**. The behavior is roughly analogous to that of the **read** function.

typein-line-form-to-eval *prompt* &optional *initial-contents initial-char-pos*

Similar to **typein-line-readline** except that the Return key does *not* terminate input — it simply moves to a new line. Only the End key terminates input. A string is returned (possibly containing Return characters).

typein-line-multi-line-read *ctl-string* &rest *args*

Combination of **typein-line-read** and **typein-line-form-to-eval**. Multiple lines are read, terminated by End, and the input is scanned by the lisp reader before being returned.

read-buffer-name *prompt default* &optional *impossible-is-ok-p*

Prints *prompt* in the typein line, and does a completing read in the mini-buffer, using the names of all the buffers as the set of possible completions. *default* is chosen if the user just types Return. The actual buffer object corresponding to the selected string is returned.

typein-line-completing-read *history default blank-line-defaults prompt alist* &optional ...

The function called by **read-buffer-name** (and many others) to do the completing read. It has more options than you even want to think about (until one of them turns out to be exactly the thing you need), all of which are described in great detail at the function definition. It allows for the use of a history as well as a default, so you can pop your way through the history to reuse earlier inputs.

completing-read-from-mini-buffer *prompt *completing-alist** &optional ...

The function called by **typein-line-completing-read** (and many others) to *really* do the completing read. You may want to call it directly because it has a somewhat simpler interface, if fewer facilities.

read-defaulted-pathname *prompt *reading-pathname-defaults** &optional ...

Used by `c-X c-F` (and many others) to read a pathname and merge it with some set of defaults.

11.5 A Real Example

Here's something taken out of the editor code, the definition for M-x Evaluate Into Buffer.

```
(DEFCOM COM-EVALUATE-INTO-BUFFER
        "Evaluates forms from the minibuffer and inserts the
results into the buffer.  You enter Lisp forms in the minibuffer,
which are evaluated when you press END.  The result of each
evaluation appears in the buffer before point.  With a numeric
argument, it also inserts any typeout that occurs during the
evaluation into the buffer." (KM)
  (LET ((FORM-STRING (TYPEIN-LINE-FORM-TO-EVAL
                       "Lisp forms to evaluate:"))
        (OUTPUT-STREAM (INTERVAL-STREAM-INTO-BP (POINT)))
        FORM)
    (WITH-INPUT-FROM-STRING (INPUT-STREAM FORM-STRING)
      (LOOP DO (CONDITION-CASE (ERROR)
                   (SETQ FORM (READ INPUT-STREAM))
                 (SYS:END-OF-FILE (RETURN DIS-TEXT))
                 (SYS:READ-ERROR (BARF "~A" ERROR)))
               (FORMAT OUTPUT-STREAM "~{~&~S~}"
                       (LET-IF *NUMERIC-ARG-P*
                               ((STANDARD-OUTPUT OUTPUT-STREAM))
                         (MULTIPLE-VALUE-LIST (EVAL FORM))))
               (MOVE-BP (POINT)
                        (FUNCALL OUTPUT-STREAM ':READ-BP))))))
```

typein-line-form-to-eval returns a string, presumably containing lisp forms. `(interval-stream-into-bp (point))` is like `(open-editor-stream :start (point) :end (point))`. We make `input-stream` point to the string of forms, and then loop, reading one form at a time from the string. For each one, assuming there are no errors, we use **format** to print the results of evaluating the form into the buffer, via the open editor stream, and move point.

11.6 Problem Set #10

Questions

1. Write **com-comment-out-lines-in-region**, and **com-uncomment-lines-in-region**, to insert (and remove) semi-colons at the beginning of each line in the region. (Both of these already exist as parts of **com-comment-out-region**, but that version includes lots of hair for handling messy cases. Write something simple.)
2. Write **com-insert-text-into-other-buffer**, which should read any amount of text from the mini-buffer (terminated by `#\End`), prompt for the name of a buffer, and append the text to the end of the given buffer.
3. Write a macro to be used either inside or outside the editor, which redirects all typeout during execution of its body to a newly created editor buffer.

Solutions

(This all goes in the zwei package.)

1.
```
(defcom com-comment-out-lines-in-region
        "Comments out each line in the region." nil
  (region-lines (start end)
    (loop for line = start then (line-next line)
          until (eq line end)
          do (insert (create-bp line 0) #\;)))
  dis-text)

(defcom com-uncomment-lines-in-region
        "Removes semi-colons from beginning of each line in
region." nil
  (region-lines (start end)
    (loop for line = start then (line-next line)
          until (eq line end)
          when (char-equal (aref line 0) #\;)
            do (let ((end-idx (string-search-not-char
                                   #\; line 1))
                     (start-bp (create-bp line 0)))
                 (delete-interval
                   start-bp
                   (if (null end-idx) (end-line start-bp)
                       (forward-char start-bp end-idx))))))
  dis-text)
```

Don't forget to add the commands to a comtab so you can use them. Do something like:

```
(set-comtab *zmacs-comtab*
            '(#\s-; com-comment-out-lines-in-region
              #\h-; com-uncomment-lines-in-region)
            (make-command-alist
              '(com-comment-out-lines-in-region
                com-uncomment-lines-in-region)))
```

2.
```
(defcom com-insert-text-into-other-buffer
             "Appends text from the mini-buffer to the end of
any buffer." nil
  (let ((text (typein-line-form-to-eval
                "Text to append to other buffer:"))
```

```
       (buffer (read-buffer-name "Buffer to append text to:"
                                 :other)))
    (with-editor-stream (str :interval buffer)
      (send str :string-out text)))
  dis-none)
```

3. Anything sent to **standard-output** during execution of *body* will be inserted into a buffer named *buffer-name*. There will also be messages inserted before and after *body* is executed.

```
(defmacro with-output-to-editor-buffer ((buffer-name)
                                        &body body)
  `(with-editor-stream (standard-output
                         :buffer-name ,buffer-name)
     (format t "~2%;;; Diverting to buffer (~\datime\)~2%")
     ,@body
     (format t "~2%;;; End of diversion (~\datime\)")))
```

Sample usage:

```
(with-output-to-editor-buffer ("test")
  (princ "The contents of the FEP file system follows:")
  (print-disk-label))
```

Chapter 12

A QUICK LOOK AT "THE NETWORK"

Although it is common to refer to a lisp machine's connections to the rest of the world as "the network," as if the machine were connected via a single mechanism to a unified system of linkages, such is not the case. There are several means of communication, operating via several different hardware and software protocols. And there is considerable overlap, with different software protocols operating simultaneously over the same hardware. It's not really very complicated, but it's easy to become highly confused if the basic issues are not kept clear.

12.1 The "Gee-Whiz" Overview

The first distinction to keep in mind is between hardware and software. The hardware basis for any given network service will be something like coaxial cable and transceiver boxes, or a serial line. The software utilizing this hardware, say "Internet" or "Chaosnet," must itself be viewed as composed of several theoretically independent levels. The various pieces of this modularity are coordinated by the namespace database. The terms used by the namespace for the different levels are *service*, *protocol*, and *medium*. (Don't be concerned if the exact meaning of the various concepts is not clear at this point; all that matters for now is that you have a general understanding of what sorts of entities the terms refer to.) The distinctions may at times appear somewhat strained or artificial, but in general a service is the highest level entity, describing what the user wants out of the network connection. A service may be discussed without reference to which network is providing it. Typical services are file transfer and remote login. Each network has its own protocol(s) for providing any particular service. Chaosnet provides file transfer service via the "qfile" protocol, while Internet provides the same service via the "tftp" protocol. A medium is a mode of connection, like "byte-stream" or "chaos"; each protocol requires some medium as the minimal type of connection which must be present for the protocol to operate. A connection can be made to a remote host in order to request a certain service when both the local and remote hosts are on a network of a type adequate to support the medium required by the protocol under which the remote host offers the desired service.

Let's look at excerpts from the namespace descriptions of two machines to see some of this terminology in action.

```
HOST JONES
SYSTEM-TYPE UNIX
MACHINE-TYPE VAX
ADDRESS CHAOS 1015
ADDRESS INTERNET 192.11.39.9
ADDRESS TAMDHU-SERIAL 3
SERVICE FILE CHAOS QFILE
SERVICE LOGIN SERIAL-PSEUDONET TTY-LOGIN
SERVICE LOGIN TCP TELNET
```

This (partly fictional) fragment states that the UNIX host "jones" has addresses on the chaos, internet and tamdhu-serial networks, and that it offers the file service (file transfers) via the qfile protocol on the chaos medium, and the login service (remote login) via either the tty-login protocol on the serial-pseudonet medium, or the telnet protocol on the tcp medium.

```
HOST TAMDHU
SYSTEM-TYPE LISPM
MACHINE-TYPE LISPM
ADDRESS CHAOS 1003
ADDRESS TAMDHU-SERIAL 0
ADDRESS INTERNET 192.11.39.5
```

And this fragment tells us that the lisp machine "tamdhu" is on the chaos, tamdhu-serial and internet networks. Combining this with the description of jones, we can tell that from tamdhu a user could invoke file transfer service on jones, since both machines are on the chaos network (and chaosnet supports the chaos medium). As for remote login service, the user gets a choice: s/he could invoke it on the network called "tamdhu-serial" (which both hosts are on, and which supports the serial-pseudonet medium) or on Internet (which supports the tcp medium).

12.2 The Beginning of the Real Explanation

Until recently, what most people had in mind when they said "the network" with reference to lisp machines was *Chaosnet*, a local area network developed in 1975 by the MIT AI Lab, specifically for use as the medium for communications among lisp machines. But as it was designed to minimize the difficulty of bringing other kinds of machines into the network, by now quite a variety of computers — as well as peripherals — may be connected via Chaosnet. The "chaos" in the name refers to the lack of centralized control.

The hardware and software portions of Chaosnet, although designed for each other, are logically independent. The Chaosnet software may operate on media other than the Chaosnet hardware, and the software for other network protocols may use the Chaosnet hardware. The transmission medium is sometimes called *ethernet*, which can be misleading because the same term is applied to a type of network (including hardware and software) developed by Xerox. But since the Ethernet hardware and the Chaosnet hardware are largely compatible, the term "Ethernet" is frequently applied to the hardware for any network using this sort of transmission medium, regardless of which software protocol is in operation.

We'll return to Chaosnet shortly, but first let's complete the overview of the lisp machine's network connections. Coexisting with Chaosnet on the ethernet cable is the *Internet*, an entirely different software protocol. The hardware requirements of the two types of network are compatible; although some kinds of computers use different boards for the two interfaces, everything external to the individual machines is identical. In the case of the lisp machine, even the internal hardware is shared, so the distinction between the two networks is solely what happens to the transmitted information in software. The software for Internet is not, however, part of the basic lisp machine system. It's another product which must be purchased separately, as "ip-tcp." The Bell Labs/Murray Hill lisp machine community has a site license, and ip-tcp is installed on all our machines. As with Chaosnet, Internet is understood by a wide variety of computers. In fact, Internet has become a Department of Defense standard, so one may reasonably expect virtually all manufacturers to support Internet in their new products, while Chaosnet may be expected to gradually fade from view.

The third means of communication available to the lisp machine is the serial i/o facility. Each lisp machine has three serial ports, which may be used to communicate with any device that understands the RS-232 protocol. At my site we connect these ports to a VAX* terminal line, to allow use of the lisp machine as a terminal logged in to a UNIX system, or to speech synthesizer boxes, or a modem. Because the related software uses a construct called a *serial-pseudonet*, it appears from the user's point of view that each lisp machine is the center of its own star-shaped network (on an equal footing with Chaosnet and Internet) connecting it to the devices at the other end of the serial lines.

So all together there are two kinds of physical connection to the outside world (ethernet and serial line), and three kinds of conceptual connection (Chaosnet and Internet via the single ethernet, and serial-pseudonet via any number of serial lines). The term "network" — now that Chaosnet no longer stands alone — usually refers to these three conceptual connections, either one of them taken individually or the set of them considered collectively. It is possible, though unusual, to define additional types of conceptual networks, sharing the existing physical connections.

Most of the physical and conceptual connections are documented in volume 9 of the Symbolics manuals, from which I have borrowed for this chapter. Chaosnet, both hardware ("ethernet") and low-level software, is covered in chapter 15. Part I of volume 9 describes the namespace database and how the namespace manages requests for network services. Serial i/o is in Part III of volume 5.

12.3 The Ethernet

The transmission medium (the *ether*) which supports both Chaosnet and Internet is 1/2 inch coaxial cable, with a *transceiver* box at each point where a machine joins the cable. A single ether must be a linear cable, with no branches or loops. The maximum length of an ether is about a kilometer. Multiple ethers may be joined by *bridges*, *i.e.*, machines on both of two ethers, which relay packets from one to the other, so that the two *subnets* may act as one large ethernet without exceeding the length limitation. For example, the portion of the Murray Hill ethernet that local lisp machine users need to be concerned with has three single ethers joined by bridges. One connects all the building 2 lisp machines (except Churchill), about five VAXen and some Sun workstations; one (the "tempo net") connects a variety

* VAX is a trademark of Digital Equipment Corporation.

of machines in building 3; the third (the "backbone") runs for about a mile (with repeaters inserted at strategic points, since a mile is more than a kilometer) connecting the other two subnets and who knows what else, hitting about fifty machines along the way. Sola (a VAX in building 2) serves as the bridge between the building 2 subnet and the backbone; Vivace links the backbone and the tempo net. The current list of subnets can be found in /etc/networks on any UNIX machine running Internet.

Returning to the operation of a single ether, one machine at a time may seize the ether and transmit a packet with the address of some other machine. The packet will be seen by every machine connected to the ether; it is up to each to check the address in the packet and decide (in hardware) whether it is the intended recipient. If the address is correct, the packet is received and relayed up to the appropriate software (Chaosnet or Internet, depending on the type of the packet). Otherwise it is ignored.

The ether can tolerate temporary breaks in the cable for about a minute — long enough to splice a transceiver in or out of the ether. The effects of longer breaks vary with the type of the machine, but can be disastrous for some — VAXen crash when reconnected after a long break. If such a machine is to be spliced out of the network while running, a pair of *terminators* should be attached to its transceiver. The terminators present an impedance similar to that of an intact ether, so the machine sees what appears to be a legitimate ether which just happens to be completely inactive. It can later be reattached with no difficulty.

Lisp machines are insensitive to ether disruptions. To physically remove a lispm from the network it is safe and easy to disconnect the transceiver cable where it plugs into the back of the machine. You may reconnect at any time. Actually, it's rarely necessary to physically disconnect the machine. The equivalent may be done in software by evaluating `(neti:reset)`, which disables the ethernet interface. `(neti:enable)` starts it up again. The CP command `Reset Network` does a reset immediately followed by an enable.

12.4 Chaosnet

The previous section discussed the portion of Chaosnet which is shared with Internet, *i.e.*, the Ethernet hardware. Now a bit about the portions which are unique to Chaosnet.

Chaosnet addresses are simple numbers, consisting of three or more octal digits. The least significant eight bits indicate the machine's address on its own subnet, or single length of ethernet cable. The higher order bits indicate which subnet the machine is on. The subnet codes used in Murray Hill are 1 for the backbone, 2 for building 2, and 3 for the tempo net. So an octal address of 406 (Pancake's address) means machine 06 on the backbone. Note that the numbering of machines on a subnet has no relation to their physical order. A lisp machine's opinion of what its own address is comes from the "Set Chaos-address" fep command. Every boot file should have one of these. The machine's opinion as to its name is derived from its address: it believes whatever the namespace database says is the mapping between names and addresses. So all that's required to make Abelour (address 1002) believe that it's really Glengarioch (address 1006) is to boot Abelour from a boot file that contains the line "Set Chaos-address 1006." No physical alterations to the network connections are involved. Then Abelour will receive and answer any packets intended for Glengarioch. There is no sense in which Abelour will not, in fact, *be* Glengarioch. (To avoid having two Glengariochs, this stunt should only be pulled when Glengarioch is down, or itself booted as somebody else.) The file "fep0:>namespace.boot" on Abelour does exactly this. We boot Abelour as Glengarioch if the real Glengarioch (our namespace server) is down, so that Abelour will act as the namespace server until Glengarioch is fixed.

The same method for address-swapping allows us to transform Laphroaig (1001) into Ghost-of-Laphroaig (401), but since the new address implies being on a different subnet, the switch additionally involves plugging the machine into a normally unused transceiver on the backbone. This trick allows us to transfer large files (like world loads) to machines on the backbone without having to go through the bridge. Executing a band transfer through a bridge takes at least twice as long as going direct, as well as completely tying up the chaos server on the bridge for the duration.

A UNIX machine's idea of who is at what address is based on the contents of its *host table*. The Chaosnet host table is in the file /usr/chaos/lib/libhosts/hosts.local. (The Internet host table is in /etc/hosts.)

From volume 9, 15.3: "The principal service provided by Chaosnet is a *connection* between two user processes. This is a full-duplex reliable packet-transmission channel. The network undertakes never to garble, lose, duplicate, or resequence the packets... When first establishing a connection, it is necessary for the two communicating processes to contact each other... One process is designated the *user*, and the other is designated the *server*. The server has some *contact name* [indicating the type of service] to which it *listens*. The user process requests its local operating system to connect it to the server, specifying the network address and contact name of the server. The local operating system sends a message (a *Request for*

Connection, or RFC) to the remote operating system, which examines the contact name and creates a connection to an existing listening process, or creates a new server process and connects to it, or rejects the request."

The first option (an existing process) is used for simple requests that can be answered quickly and easily, such as a request for the current time. The main server process handles these itself. More elaborate requests, which require extended attention and multiple interchange of packets, are handled by spawning a new process for that purpose. On a UNIX chaos server, this means simply executing a file in the /usr/chaos/server directory. The name of the file will be exactly the contact name that was used to request the connection. So, for example, if a lisp machine wants to read a file from a UNIX machine, it would send an RFC packet to the UNIX machine, with contact name "FILE." The chaos server on the UNIX end, upon receiving the packet, would start up a process running the contents of the file /usr/chaos/server/FILE. The chaos server would then route any further packets related to the file transfer to this new process. The effect is as though the FILE process on the UNIX system and the process requesting the file transfer on the lisp machine were communicating directly with each other.

From a lisp machine, the easiest way to establish a connection with a server process on a remote host is with the function **chaos:open-stream**. The two required arguments specify the host and the contact name. (Several optional keyword arguments offer more detailed control of the type of connection.) If a connection is successfully established, **chaos:open-stream** returns an open stream object. You may then use all the usual messages to read characters from and write characters to the stream. Conversion to and from the packet level is completely transparent — all the user sees is a character stream.

12.5 A Bit More on Serial Streams

The programmer interface to the serial i/o facility also works via lisp stream objects. The function **si:make-serial-stream** returns an open stream to one of the serial ports (which one may be specified with the `:unit` keyword argument). The usual messages for reading and writing may again be used. Details are in chapters 21 and 22 of volume 5.

12.6 The Role of the Namespace

The namespace database coordinates interaction with the various network facilities. It is always possible to directly manipulate the stream connections as outlined above, but this is necessary only if you are adding a new type of network service. To invoke any existing service which has already been integrated into the namespace, you may use the higher-level interface provided by the namespace. This interface insulates the user from having to know any details about how the connection is established, or even which network is used. The namespace takes care of finding an appropriate path.

The simplest way to invoke a network service through the namespace is with the function **net:invoke-service-on-host**. The first argument is the name of a defined *service*, and the second is a host object. The local host will find the best path to the target host, over all available network connections, and use it to invoke the named service. So to obtain Sola's idea of what time it is, I could do `(net:invoke-service-on-host :time (si:parse-host 's))`. (Seeing as how Sola's namespace entry says that it provides the `time` service via the `time-simple` protocol on the `chaos-simple` medium, we can tell this will end up being a Chaosnet transaction, but we needn't be concerned with that level of detail.)

In cases where you need some service but don't care on which host it is invoked, the function **net:find-paths-to-service** (called with an argument of the service name) returns a list of *service access paths*, one for each host which provides the service. The list is sorted by decreasing desirability. Having decided which path to use, you may invoke the service by calling **net:invoke-service-access-path** on the chosen service access path.

Another option is to invoke a service simultaneously on more than one host, with *service futures*. You can then pick the first or best of several responses without a long waiting period. The handiest way to manipulate service futures is with the macro **net:invoke-multiple-services**. This example prints every host's idea of the current time, ignoring those which fail to respond:

```
(defun all-hosts-time ()
  (net:invoke-multiple-services
    ((net:find-paths-to-service :time) (* 60 10) "Time")
    (host time)
    (sys:network-error nil)        ; catch the error and do nothing
    (:no-error
     (format t "~&~A: ~:[unknown~;~\TIME\~]" host time time))))
```

12.7 Troubleshooting

Typing Function-H [or evaluating `(hostat)`] will show which machines your lispm is able to contact via Chaosnet. If you can't reach any remote hosts, something is probably wrong locally. If you can reach some and not others, something is probably wrong out in the network somewhere. Here are a couple of hints:

Don't forget about **neti:reset** and **neti:enable**. If a reset has been done, **hostat** will fail everywhere.

The transceiver cable locks into place, at both the machine and transceiver ends, with a tab which is slid sideways after the connector is pushed into the socket. A fairly common cause of "network failure" is for the cable to fall off the back of the machine because the tab wasn't used (or was defective).

Check the state of the process called "3600 Ethernet Receiver" in Peek. If it's not "Ethernet Packet," something's wrong — maybe you ignored a Function-0-S notification.

For more elaborate troubleshooting, try the "network" option in Peek.

APPENDIX: BASIC ZMACS COMMANDS

While it is true that there are a great many Zmacs commands, and trying to learn them all would be a close to hopeless task, it is also true that the situation is really much less difficult than it might at first appear. For one thing, you can edit files quite effectively with a relatively small subset of the Zmacs commands. (This should not be taken to imply that the remaining commands are superfluous; the "quite effective" editing you can do without them is transformed into astonishingly effective editing with them.) Another saving grace is that there is some pattern to the pairing of keystrokes with editing functions. For instance, *control* characters often act on single letters or lines; *meta* characters on words, sentences, or paragraphs; and *control-meta* characters on lisp expressions. Thus c-F moves forward one character, m-F moves forward one word, and c-m-F moves forward one lisp expression. c-K means "kill" (delete) to the end of the line, m-K means kill to the end of the sentence, and c-m-K means kill to the end of the current lisp expression. So the amount of memorizing you have to do to start editing is really not very great.

I can't overemphasize the utility of the Help facility in Zmacs. It can be a real lifesaver, both when you don't know what commands there are to do something, and when you've forgotten how to invoke a command you know about. So don't limit yourself to the commands listed below. Consider the list a crutch, to help get you started, but try to leave it behind as soon as possible.

Movement Commands

`c-F`	Move forward one character
`c-B`	Move backward one character
`c-N`	Move down one line ("next")
`c-P`	Move up one line ("previous")
`c-A`	Move to the beginning of the line
`c-E`	Move to the end of the line
`mouse left`	Move to mouse position
`m-F`	Move forward one word
`m-B`	Move backward one word
`m-A`	Move to the beginning of the sentence
`m-E`	Move to the end of the sentence
`m-[`	Move to the beginning of the paragraph
`m-]`	Move to the end of the paragraph
`m-<`	Move to the beginning of the buffer
`m->`	Move to the end of the buffer
`c-m-F`	Move forward one lisp expression
`c-m-B`	Move backward one lisp expression
`c-m-A`, `c-m-[`	Move to the beginning of the current definition
`c-m-E`, `c-m-]`	Move to the end of the current definition

Deletion Commands

`c-D`	Delete forward one character
`Rubout`	Delete backward one character
`Clear Input`	Delete to the beginning of the line
`c-K`	Delete to the end of the line
`m-D`	Delete forward one word
`m-Rubout`	Delete backward one word
`m-K`	Delete forward one sentence
`c-m-K`	Delete forward one lisp expression
`c-m-Rubout`	Delete backward one lisp expression

`c-Y`	Restore ("yank") text deleted with any of the above, except c-D and Rubout
`m-Y`	Immediately following a `c-Y` or another `m-Y`, replace the yanked text with the previous element of the kill history

Region Commands

`c-Space`	Set the mark at the current position, and turn on the "region." Subsequent movement commands will define the region to be the area between the mark and the new position.
`c-W`	Delete the region, putting it on the kill history
`m-W`	Put the region on the kill history without deleting it
`mouse middle`	Mark (make into the region) the object the mouse is pointing at
`mouse drag left`	Mark the area dragged over (between button press and button release)

File Commands

`c-X c-F`	Read ("find") a file into its own buffer, creating an empty buffer if the file doesn't exist
`c-X c-S`	Write ("save") a buffer back to its file
`c-X c-W`	Write a buffer to any file, specified by name
`M-x Compile File`	Compile a file, *i.e.*, write a binary version of the file. This has no effect on the current Lisp environment. (Compare to `M-x Compile Buffer`.)

Buffer Commands

`c-m-L`	Switch to the previous ("last") buffer
`c-X B`	Switch to a buffer specified by name
`c-X c-B`	Display a list of the buffers

Lisp Commands

`c-sh-E`	Evaluate (call the Lisp interpreter on) the region, if

	it's active, or the current definition if it's not
`c-sh-C`	Compile the region or current definition into the Lisp environment
`M-x Evaluate Buffer`	Evaluate the entire buffer
`M-x Compile Buffer`	Compile the entire buffer into the environment. This has no effect on the file system. (Compare to `M-x Compile File`.)
`End`, `s-E`	*Murray Hill Standard Utilities only.* Evaluate the region or current definition and insert the result into the buffer.

Miscellaneous

`Suspend`	Enter the typeout window. (`Resume` returns.)
`m-.`	Find the definition of a given function
`c-X D`	Directory edit — shows a directory listing and enables manipulation of the files in it
`Help A`	Apropos — list all commands containing a given substring
`Help C`	Describe the command associated with a given keystroke
`Help D`	Describe a command specified by name

INDEX

(please clip here)

Please send me a copy of the examples tape that accompanies *Using the Lisp Machine*. Enclosed is my check for $40.00, plus applicable sales tax,* made payable to Symbolics, Inc. This fee includes domestic postage and handling. (Outside North America, add $10.00 for postage.)

Ship the tape to the following street address (no PO boxes, please):

(name)

(company)

(street)

(city)　　　　*(state)*　　　　*(zip)*

(　　)

(phone)

*Residents of the following states add appropriate sales tax: AZ, CA, CO, CT, FL, GA, IL, KS, MA, MN, NJ, NM, NY, OH, PA, TX, VA, WA.

LISP *LORE: A GUIDE TO PROGRAMMING THE LISP MACHINE is a course in programming a Lisp machine. The book presents a readily understandable introduction to several representative areas of interest, including enough information to show how easy it is to build useful programs on the Lisp machine. A graduated series of exercises, with hints and solutions, is also included in the text.*

The book assumes some background in LISP and experience with some dialect of the language; however, no experience with the Lisp machine itself is required.

Kluwer Academic Publishers *0-89838-220-3*