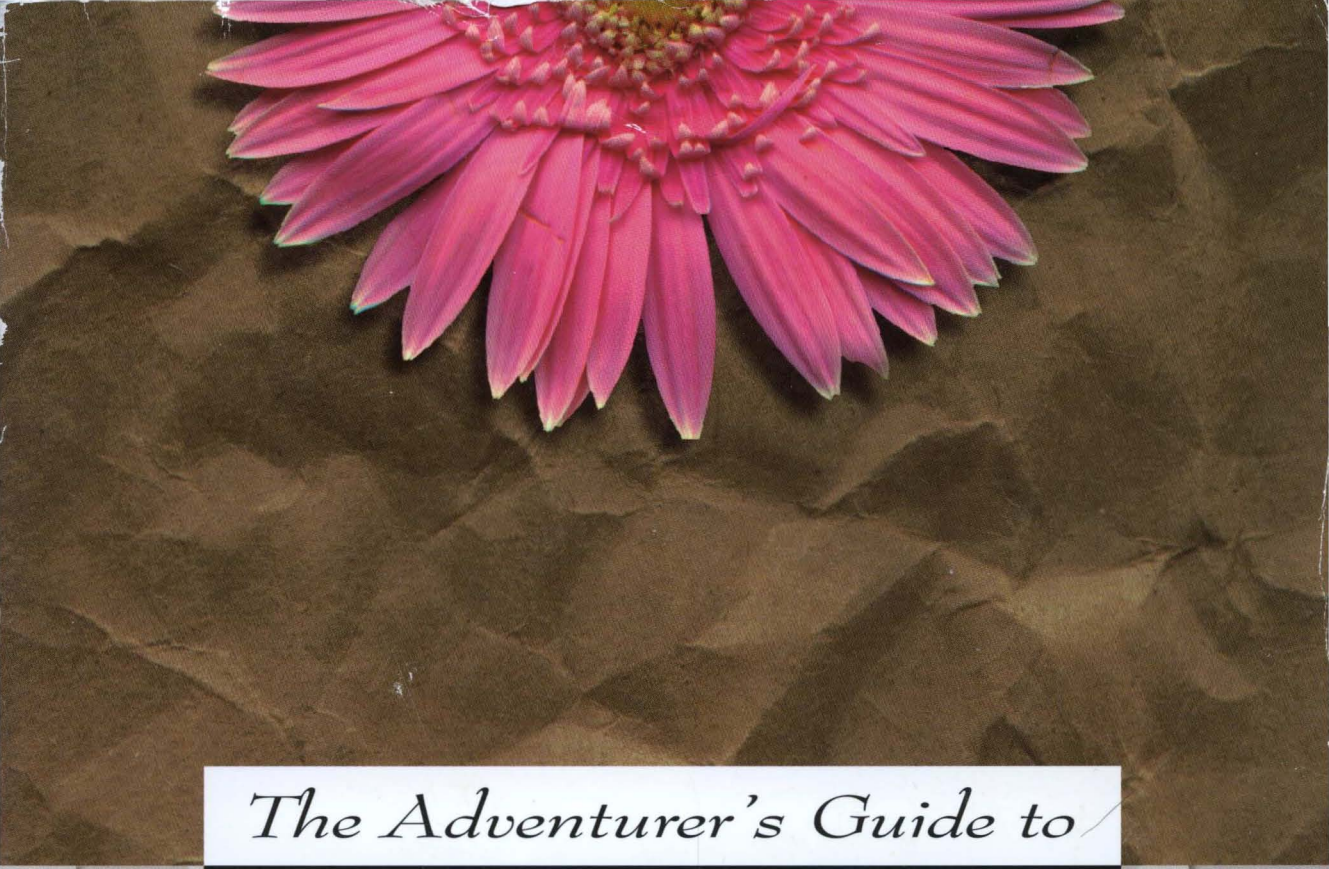


The Adventurer's Guide to

INTERLEAF LISP

David Weinberger

ONWORD PRESS

Adventurer's Guide to Interleaf Lisp

David Weinberger

ADVENTURER'S GUIDE TO INTERLEAF LISP
By David Weinberger

Published by:

OnWord Press
2530 Camino Entrada
Santa Fe, NM 87505-4835 USA

First Edition, 1994
SAN 694-0269
10 9 8 7 6 5 4 3 2 1
Printed in the United States of America

Acquisitions Editor: David Talbott
Project Editor: Margaret Burns
Production Manager: Carol Leyba
Production Editor: Sandra McDougle
Cover Design: Lynne Egensteiner
Copy Editor: Mary Margaret McCormick

Library of Congress Cataloging-in-Publication Data

Weinberger, David
Adventurer's Guide to Interleaf Lisp

Includes index.

1. Interleaf Lisp (computer programming language) 2. Desktop publishing I. Title
93-85301
ISBN 1-56690-042-5

Trademarks

Warning and Disclaimer

This book is designed to provide information about Interleaf Lisp. Every effort has been made to make this book complete and as accurate as possible; however, no warranty or fitness is implied. The information is provided on an "as-is" basis. The author and OnWord Press shall have neither liability nor responsibility to any person or entity with respect to any loss or damages in connection with or rising from the information contained in this book. This book was prepared using Interleaf 5 and Interleaf 5 for DOS.

About the Author

David Weinberger has been working with Interleaf tools for seven years. He has written for *Byte, PC Magazine, UNIX World, Computer World, The New York Times, Smithsonian Magazine,* and *TV Guide*. He speaks annually at national and international ICON.

Thanks for the Help

I'd like to thank the following people for their generosity of time and expertise: Tim Anderson, Tom Borgman, Keith Corbett, Don Davis, Stacy and George Dymalski, Paul English, Tim Giebelhaus, Kimbo Mundy, Gary Wasserman and Information Technology Partners, Inc. And most of all, I owe thanks to my wife, Ann Geller, for giving me all the nights and weekends this book has required.

Cover Art

Cover design by Lynne Egensteiner, using QuarkXpress 3.1 and Adobe Photoshop 2.5.1.

OnWord Press

OnWord Press is dedicated to the fine art of practical documentation. In addition to the author who developed the material for this book, other members of the OnWord Press team make the book end up in your hands.

Thanks to Dan Raker, David Talbott, Carol Leyba, Gary Lange, Margaret Burns, Lynne Egensteiner, Tierney Tully, Michael Hadley, Catherine Hemenway, and Mary Margaret McCormick.

Installing the Bonus Lispware Disk

This disk contains files "packed" together into one file by the Interleaf Desktop Utility (IDU). The IDU file is named "leafware.idu". To use these files, you must unpack them using IDU. When IDU unpacks them, you will end up with a subdirectory named "lisp-ware" that should appear as an Interleaf desktop cabinet. Inside will be further cabinets, as well as README files. IDU will ensure that the files you receive follow the differing naming conventions used by MS-DOS, UNIX, and other operating systems supported.

1. To unpack the files with IDU, first locate the idu utility. On UNIX systems, its probable path is: /interleaf/ileaf5/bin/idu. On DOS, its probable path is: \ileaf5\bin\idu.exe.
2. Next, change to your desktop directory (e.g., "cd D:\desktop" in DOS).
3. Now extract the files by typing at the prompt:

   idu-path -xf idu-file-path

   where "idu-path" is the path to your idu utility and idu-file-path is the path to "lisp–ware.idu."

   For example, in DOS the command might be:

   \ileaf5\bin\idu -xf A:\lispware.idu

   and in UNIX it might be:

   /interleaf/ileaf5/bin/idu -xf /pcfs/lispware.idu
4. If you have Interleaf up and running already, do a "Rescan" on your desktop to force it to notice the new files put there.
5. If you have copied "lispware.idu" to your hard drive, after "unpacking" it you may delete it.

Contents

Chapter 1
Introduction

Interleaf 5™ is an amazingly powerful publishing product. From the end user's point of view, it can do just about anything imaginable to text and graphics.

To the developer, however, that's just the starting point.

Interleaf 5 contains — in fact, much of it is built with an extension language which allows a programmer to take advantage of all of Interleaf 5's text and graphics document functionality. The programmer can use this extension language — Interleaf Lisp™ — to modify Interleaf 5 or to build entirely new applications out of its pieces.

For example, a developer could use Interleaf Lisp to

- Add new functionality to Interleaf 5
- Alter Interleaf 5's interface
- Build *active documents*, programmed with the "intelligence" to access information, evaluate it, and act on it
- Create links to other data
- Automate the document work flow
- Create entirely new applications that use Interleaf 5 as, for example, an interface or display engine.

All this is possible because the Interleaf Lisp development environment is so powerful. It was not designed for casual end users; Interleaf built the development environment with professionals in mind. Nevertheless, with some guidance and some patience, those of us who are not professional programmers can begin to use the tools that are sitting in Interleaf 5 to take advantage of the product's radical *re-programmability.*

That's what this book is about.

Why is this a good environment?

Interleaf Lisp is part of a highly productive development environment. That's not simply because the language itself is useful. It's primarily because the environment is *document-object–oriented.*

In practice, this means that Interleaf has already created a set of very powerful objects. For example, Interleaf tables already know how to size themselves to their contents, paginating themselves if necessary. This sort of functionality would be difficult to build from scratch. But because Interleaf has already built these tables, the developer can use them without doing any further work. For example, if you wanted to build an application that puts data into a particular cell in a table, all you have to do is tell the table cell (actually, a marker in the table cell's microdocument) to accept the contents. That is a single line of code. Everything else that happens — perhaps the table resizes, repaginates, causes a ripple through a 1,000-page document that involves updating hundreds of autonumbers and references — happens automatically, without the programmer having to worry about it.

In addition, Interleaf has made available a set of tools for the developer — an editor, debugger, compiler, on-line help, etc. — that lets a developer create code very productively. (This book does not assume your version of Interleaf 5 came with these tools; while they greatly aid you, they are not an absolute requirement.)

But the real test of the value of a development environment is how well developers do with it. I believe you'll find Interleaf's environment not only productive, but tremendous fun.

This book will help you give it a try.

Who is this book for?

Interleaf has a variety of programs set up to help professional developers become proficient at working with Interleaf Lisp. This book is not intended primarily for them.

Most of Interleaf 5's users will find that the active documents that come with the system are more than enough for them. They can get all the configurability and extensibility they need by using what comes in the box. This book is not intended primarily for them, although they might find it an inducement to continue to the next level of expertise.

Then there is a set of users who are ready to push the envelope. They are not professional developers, but share some of the following characteristics:

- Power users of Interleaf 5
- Ask themselves not only *how* to use a feature but also *why* it was designed the way it was
- Have looked at pieces of Interleaf 5 through their operating system, e.g., have used their OS's file browser to look at the various "invisible" backup files Interleaf creates.

Such a person is likely to fall in love with Interleaf Lisp as soon as she or he encounters it. This book is for that person.

There are some prerequisites:

- You should have some experience programming in any language, even as a hobbyist; this book will not teach you the basics of computers or of programming.
- You should have a good sense of how Interleaf 5 works; this book will not teach you the basics of Interleaf 5.

If you meet those two criteria, you should be able to use this book to good advantage.

Risks

Interleaf Lisp gives you a tremendous amount of power. It also gives you, therefore, the chance to screw up royally.

There are two fundamental things you can screw up:

- You can unintentionally destroy documents and other data.
- You can introduce "stuff" into your system that will do weird and sometimes not–so–wonderful things to Interleaf 5's processes. For example, you may discovered that you can no longer type into documents, or that all your menus have gone away, because of unintended side-effects of a script.

The safeguard against the first sort of screw-up is to back up your data and documents frequently, and to never have any documents that you care about open on your desktop while you are experimenting with Interleaf Lisp. (You can do damage to unopened documents as well; it's just easier with opened ones.)

The safeguard against the second sort of screw-up is not to do any work with Interleaf Lisp and not to accept any active documents onto your desktop.

If you decide to go ahead anyway, if your system starts acting strangely, exit and reload. If it's still acting strangely, check your *Profile* drawer and the clipboard to see if there are any active documents in there that may be getting loaded when Interleaf 5 loads. Finally, be patient and do not call Interleaf customer support unless you are quite, quite, quite sure that the problem is with the software as delivered by Interleaf, and is not due to one of your Interleaf Lisp experiments.

Design philosophy

There are, of course, many ways to write an introduction to a programming environment. Given who this book is for, I have taken a very particular approach.

- The aim of the book is not to give you an abstract knowledge of the Interleaf Lisp functions available to you but to help you design and create small applications and utilities. When quick-and-dirty is easier to understand and just as good in a small program as slow-and-clean, I've gone for quick-and-dirty every time. This book does *not* aim at purity of Lisp expression. (I'm an Interleaf Lisp hobbyist and fan, not a programming professional.)
- This is not a comprehensive reference work. There are areas and functions I have skipped because I consider them to be advanced topics. Within any one area, I have not attempted to provide comprehensive documentation, for the same reason.
- While I have tried to make the various functions easily accessible, as in a reference work, the book often assumes you've read at least somewhat sequentially. For example, the section on working within a document occasionally assumes you've read the prior section on working on the desktop.
- The book is structured in general in the order in which a fledgling programmer needs questions answered. So, for example, a section on working with components will begin with an explanation of how the programmer accesses components in the first place. That discussion may not be first in terms of alphabetical order, but it is first if you're in the process of building an application.
- I have not been a slave to consistency. For example, most functions have an example of how to invoke them. Some do not because it's so obvious or because, for some other reason, it didn't seem appropriate.
- I have tried to include lots of sample functions. When possible, I've tried to make them useful. But, since their primary aim is to illustrate points, I have sometimes included some less-than-useful programs.
- Because the sample programs are meant to illustrate points, I have tended to use the most explicit, most easily understood constructions, rather than the most compressed, fastest or most elegant. Professionals may look at my code and snicker. But the rest of us, I hope, will find it easier to make sense of what's going on in the samples.
- I have used my own experience as a devoted hobbyist as a guide. This book is, therefore, a somewhat idiosyncratic tour of Interleaf Lisp. I have tried to hit the major points, but the fact that there is nothing in here on, for example, the equation editor, is in part a result of the fact that I don't know anything about building equations. In fact, however, all books like this suffer from the same risk, except when they say that they skipped a topic because it is "advanced," they don't admit that "It's advanced" is really often a way of saying, "I don't care about it."

Other tools

This book will get you on the road. But there are other routes for you to take instead of or in addition to this one.

- Interleaf User Groups are a great way to get information — and the annual ICON user group convention can be a gold mine.
- Interleaf training.
- The Interleaf Developers Toolkit will give you the official documentation as well as a powerful set of development tools.
- Interleaf's electronic bulletin board, LeafLine, is a source of additional scripts.
- Interleaf's user dicussion group or the Internet can be accessed at the address comp.text.interleaf.

Note on the user interface

This book assumes you are using the traditional Interleaf user interface, not the Motif, Open Look, Macintosh, or Microsoft Windows versions. Much of what this book teaches you should transfer directly to Interleaf 5 running under those interfaces, except for some big chunks of the user interface. For example, while Interleaf stickups will probably be supported in those new user interfaces, you can be just about certain that property sheets will not be. So, if you are planning on moving to those new interfaces, you should be careful to isolate and identify the parts of your code that address the user interface, and you should stay away from property sheets and popups.

Disclaimer

I certainly do not claim that any piece of code in this book is written optimally. And while they've all been tried out, it's always possible that they won't work for you because you typed it in wrong, the Interleaf Lisp changed since this was written, there is something screwy in your environment that keeps it from working, there was something screwy in my environment that allowed it to work, or I made a mistake. (I haven't gotten around to becoming infallible yet.)

Remember, if things go wrong because of what you've done with Interleaf Lisp, do *not* call Interleaf Support. They can't possibly debug your programs for you, and Interleaf, Inc. obviously can't be held responsible for any of the damage you may do inadvertently while adventuring with Interleaf Lisp.

Chapter 2
Getting Started with Interleaf Lisp

Interleaf Lisp (I-Lisp) is based on a version of Lisp called "Common Lisp." But I-Lisp diverges from Lisp wherever necessary to make it work better in Interleaf's unique active document environment. In this chapter, you'll learn the basics of programming in I-Lisp.

I am assuming that you have used at least one programming language before, whether it's BASIC, Pascal, C, Fortran, etc. So I will not explain common programming basics such as the nature of variables or of loops; if you're not familiar with such concepts, there are many books that can get you started.

I-Lisp, like the Lisp it's based on, makes heavy use of the concept of *lists*; in fact, "Lisp" stands for *list processing*. You will soon get used to working with this data structure. In addition — and perhaps harder to grasp — Lisp is based around the notion of *functions*; once you're used to them, you'll find it hard to believe you ever did without them.

Functions

A function is a piece of code (perhaps a single line) that carries out a set of instructions and *returns* a value. (The value can be a single number, a list or some other data structure.) In Lisp, you conceive of your program as a set of functions which are evaluated; the word "evaluated" is used instead of "executed" to remind us that functions return *values*.

Many functions are built into I-Lisp. For example, there are functions for taking a string of characters and returning a particular letter in it, for taking two numbers and returning the lower of the two, for taking a number and returning its square root. There are also functions peculiarly useful within an Interleaf document, such as functions for inserting text anywhere in a document, for telling you exactly how many components into a document you are, and for transforming the shape of a graphic.

Lisp is an extensible language, which means you can add your own functions. For example, if as part of a program you need to look at the first character of every component in a document, you might create a function that will do that. You give it a name, tell it what information it's going to need when it runs (e.g., the name of the document it looks in), and write the function. From then on, you have it available to you.

You mark some code as a function by using the word *defun* (as in "define function"). Here's what one looks like:

2–1 open–document

```
(defun open-document (doc)
  (tell doc mid:open))
```

This function is named *open–document*. It takes one *argument*; an argument is information coming into the function. In this case, the argument coming in is put into the variable *doc*. On the second line (we could have done this in one big line; I-Lisp doesn't care) is the command to be evaluated. In this case, it tells the document to open, as we'll discuss later.

If you put this into a script, exactly nothing would happen. That's because it's a *function*, which must be invoked as you would invoke a subroutine or procedure. A function is a tool in your tool chest. Tools don't do anything until they're used.

To *use* a function, you write a line that *calls* or invokes the function, causing it to be *evaluated* (or, *eval*'ed). For example, the line *(open–document data)* would cause the *open–doc* function to execute, passing it the document for which *data* is a variable. (See Chapter 4 to learn how to actually try out scripts and functions.)

Of course, if you were to start up Interleaf 5 and execute a script that simply had the line *(open–document data)* in it, you'd get an error message because the software wouldn't know that the *open–document* function existed or where to find it. (It also wouldn't know the contents of the variable *data*.) You have to *load* the functions you need, or tell the software where to look when an unknown function is called. I-Lisp provides a variety of methods for doing this, which we'll discuss later.

Let's look at another example. Suppose there is some calculation you need to do frequently in a program. It might be figuring mortgage payments or deriving cube roots. For our example, let's say you frequently need to multiply two numbers and add a one. Rather than rewriting the same lines of code every place you need to do the calculation in your program, you would write a function such as the following:

2–2 times–2–plus–1

```
(defun times-2-plus-1 (n1 n2)
  (+ 1 (* n1 n2)))
```

To *use* this function (named *times–2–plus–1*), you would use an expression such as: *(times–2–plus–1 23 106.7)*. You'd type this into a Lisp icon, save it, select the icon

and do a Custom–>Load on the desktop. This would invoke the function and send it the two numbers (in this case, 23 and 106.7). When the function executes, it will substitute 23 for *n1* and 106.7 for *n2* and give you the right answer.

An I-Lisp program usually consists of some set of functions and some lines of code that cause those functions to be executed. The lines that cause the functions to execute may either do so immediately upon running the program, or they may tie the execution of the functions to some other event. For example, the program might cause the *times–2–plus–1* to execute whenever a user presses a particular key or makes a particular menu choice.

Data types

I-Lisp understands several different sorts of data. You can mix data types easily, and convert from one to another. You can even take something that acts as data in one context and have it act as a program in another.

The basic data types are:

- *Numbers.* Numbers can be integers (whole numbers) or real numbers (which include a decimal point and some numbers to the right).
- *Characters.* Characters are integers that are taken as code numbers for a character that can show up on the screen. For example, 65 is the code for the character "A." There are functions in I-Lisp for interpreting numbers as the characters they code and vice versa.
- *Strings.* A string is a sequential collection of characters, such as "this is a string."
- *Lists.* Lists are sequences of data. A list can contain data of more than one type. Lists can even contain other lists. Because lists are fundamental to the way I-Lisp thinks, we'll discuss them in detail below.
- *Array.* An array consists of a set of data, all of the same type, that user can access by number. Lists and arrays are similar but there are different ways of getting data into and out of lists and arrays. (This book will not discuss arrays.)
- *Symbols.* A symbol is a collection of characters (any so long as it includes at least one character that isn't a digit) that stands for something else. A symbol typically is used as a variable (to hold some value, such as a number or a string), as a name of a function, or as a *keyword* (a group of words with special, reserved meanings to I-Lisp).

I-Lisp also allows you to define your own compound data types by declaring a *structure*, but that is one of the advanced topics not covered in this manual.

Objects

Interleaf 5 is an *object-oriented* system. Object-oriented systems consist of . . . objects. An object, as opposed to a traditional data structure, not only has data but also has things it

knows how to do. So you can ask an object to open itself, or to cut itself, or to move itself. And if it has been programmed to know how to do these things, it will.

Interleaf objects are things like: books, folders, cabinets, documents, components, frames, graphic shapes and groups, photographic images, microdocuments, inline components, charts, tables, table rows, and table cells.

Interleaf objects know how to do things to themselves such as: open, close, move, delete, capitalize, rotate, fill, adjust contrast, shear, spellcheck, hyphenate. Of course, not every sort of object can do each of these things. For example, text components know how to adjust their line spacing, but graphic shapes don't. And while both documents and folders know how to open themselves, "open" means something different to each of them. The operations objects know how to perform are called *methods* in object-oriented parlance.

I-Lisp allows you to create new types (or *classes*) of objects with their own methods and to change the methods of already-existing classes. For example, documents come with a close method (i.e., they know how to close themselves). But suppose you want documents to do something different when you tell them to close; suppose you want them to encrypt themselves. Interleaf I-Lisp allows you to change the close method of documents. (You might want then to change their open method so that they automatically decrypt themselves.)

Objects also have *properties*. A property is some information about the object. For example, a component object not only has a method for changing its margins, it also has left and right margin information as properties. When you change the component's margins (either by using the component property sheet, or by using an I-Lisp program), you are changing the component's properties. (Technically, a property is just more data about an object. But in a document system such as Interleaf, it is useful to distinguish content data from property data.)

The predefined objects Interleaf 5 comes with have predefined properties. For example, folder objects have the properties of *name*, *position-on-desktop*, etc. Graphic frames have the properties of *name*, *type-of-anchor*, *height*, *width*, etc. You can always find out what properties an object has. You can alter the contents, or add your own category of properties to objects.

Classes

Objects are always instances of a class. A class contains the set of methods that each object inherits. So, if you create a new object of the class *doc–cmpn–class* — that is, a new document component — it will automatically know how to do the things that other components know how to do. It *inherits* its methods from its parent.

You can create new classes of objects. For example, you might want to create a class of document that knows to check a security list before opening itself, or a class of diagram-

ming object that knows to perform an action if it is selected (i.e., a button), or a class of components that know how to do math.

Interleaf comes with a rich set of classes already defined. Here are some examples:

Class	Example Objects	Example Methods
dt–book–class	book	open, close, create index of its contents, print its contents
doc–document–class	document	open, close, rename, set page size
doc–cmpn–class	component	join, split, set line-spacing
doc–table–class	table in a document	accept a fill pattern
doc–page–class	page	have a number
dg–ellipse–class	ellipse drawn in a frame	rotate, fill, move
doc–frame–class	frame	get properties, size, have shared contents
dg–micro–doc–class	microdocument	join, split, change fonts
doc–cmpn–editor–class	internal editor that knows how to manipulate components	find a component, change a component's properties

These are just samples. The list of predefined classes is quite extensive. And, of course, you can always add to it.

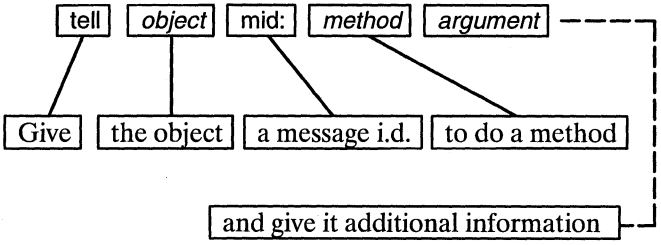
Sending messages to objects

With I-Lisp you get objects to do things by asking them politely. If an object has a *method* for doing something — or if you supply it with a method — the object will know how to respond and will take the appropriate action. (If it doesn't have a method, it will look to its parent class. If the parent class has a method, the object will use its method. Otherwise it looks to the parent's parent, etc. If it finds no method anywhere, it gives you an error message.) For example, if you tell a document to open itself, it will create a document window, lay out all the text and graphics in the document, and show itself in the window. If you ask a folder to open itself, it will create a directory window and show its contents as icons. And if you ask a component to open itself, it will tell you that you're making a mistake because components don't have a *method* for opening themselves, unless you've supplied one.

The command that sends a message to an object is quite simple in format:

(tell object mid:method argument)

This can be translated as:



The *argument* line is dotted because not all methods require additional information. Some always do, some sometimes do, and some never do. Some require a *keyword* before the argument to tell it what type of argument it is. A keyword is a word with a special meaning in Interleaf Lisp, which by convention begins with a colon (e.g., *:line–spacing, :begin–new–page, :hide–side–bar).*

How do you indicate which object you want to send a message to? You can't just use its name because a name is a *property* of an object, not an object itself; once you have an object, you can *tell* it to *mid:get–name* (*get–name* is a method for named objects), and it will tell you its name. So you need some other way of indicating what the object is.

There are several ways of indicating an object. For example, the expression *(doc–current–icon)* points at whatever icon is currently in use. And *(doc–point–cmpn)* indicates the component your text caret is currently in. So the expression *(tell (doc–current–icon) mid:get–name)* would give you the name of the current icon; and *(tell (doc–point–cmpn) mid:get–name)* would give you the name of the component the text caret is in. Later you'll learn other ways to indicate other objects.

Suppose, for example, that you want to set the current component's line spacing. Components have methods for setting line spacing, of course. If you *tell* a component to set its line spacing, it needs to know what you want to set it to. So, the expression *(tell (doc–point–cmpn) mid:set–props :line–spacing)* wouldn't work, but *(tell (doc–point–cmpn) mid:set–props :line–spacing 100000)* would. (Interleaf likes measurements expressed in rsu's — which stands for *ridiculously small units.* There are 1,228,800 rsu's to the inch.)

Return

We just said that *(tell (doc–current–icon) mid:get–name)* would tell you the name of the icon currently in use. But just typing in those letters will do nothing; you have to type them in an environment that will let you run them. That means the built-in interpreter has to

be told to read the letters, figure out that they are an I-Lisp instruction, and then execute the instruction.

But even once you learn how to do that, executing the instruction will have no visible effect. Likewise, the I-Lisp expression *(+ 1 2)* is an instruction to add 1 and 2. But executing this instruction will do nothing visible.

What happens when you execute *(+ 1 2)* ? The I-Lisp interpreter *returns* the right answer. But you haven't told I-Lisp to do anything with the answer that it has returned.

There are lots of things you might do with the answer. You might want it displayed in a stickup. You might want it to be multiplied by 6. You might want it to be stuck into the cell of a table. None of these things will happen unless you explicitly tell I-Lisp to do one of them. You will soon learn how to tell I-Lisp to do something with the information it returns to you.

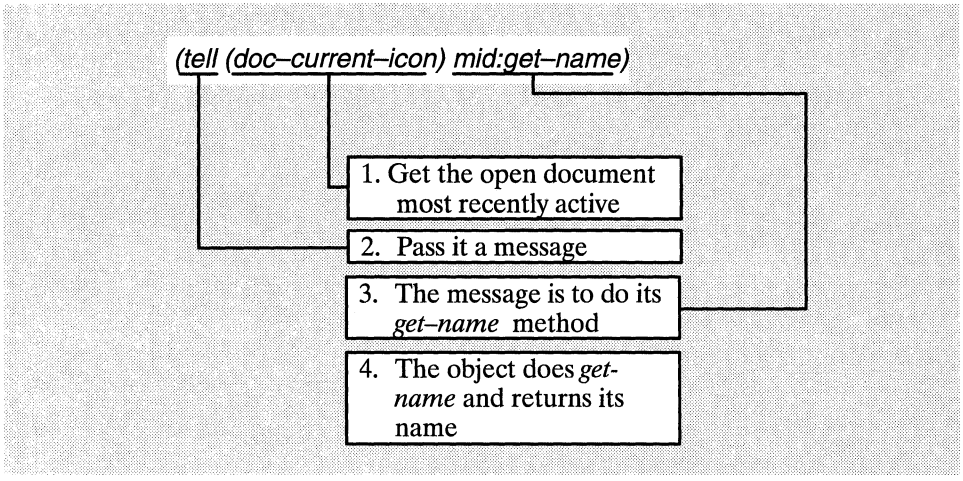
The key point in this section, however, is that there is a return for every instruction, even if the return is just the fact that nothing is returned.

Sometimes the return is obvious. For example, the return of *(+ 1 2)*, which adds 1 and 2, is the result of the math that I-Lisp does. The return of *(tell (doc-current-icon) mid:get-name)* is the name of the icon. But what's the return of *(tell (doc-point-cmpn) mid:set-props :line-spacing 100000)*, which sets a component's line-spacing? After all, you're not telling the component to adjust its line spacing in order to get some information; you're telling it that in order to adjust the line-spacing. Nevertheless, every instruction or function returns something. In the case of the line-spacing command, I-Lisp returns *t* (which stands for *true*) if the command worked, and *nil* (i.e., *nothing*) if for some reason the command couldn't be executed.

In fact, if you look at the instruction *(tell (doc-current-icon) mid:get-name)* you'll see that it takes advantage of the fact that instructions return information. The expression *(doc-current-icon)* is itself an instruction that gets evaluated by the I-Lisp interpreter. It returns the object that is the current icon. In fact, when the I-Lisp interpreter looks at *(tell (doc-current-icon) mid:get-name)*, it substitutes what *(doc-current-icon)* returns for "*(doc-current-icon)*" itself.

Here's what happens step by step:

- The *tell* flags that the next piece of the instruction is the object the message is going to.
- The *(doc–current–icon)* gets executed and returns the object currently in use. This is now taken as the object receiving the message.
- The *mid:* flags that the next phrase is the method to be run.
- The *get–name* is the method.



Likewise, you could substitute the expression *(+ 4000 6000)* for the number 10000 in *(tell (doc–point–cmpn) mid:set–props :line–spacing 10000)* without changing the outcome, for I-Lisp would add 4000 and 6000, and return 10000.

t or nil

There are two special values in I-Lisp that are frequently the returns of various functions and expressions. We mentioned them briefly in the previous section, but they warrant a small section to themselves.

The returns are *t* (true) and *nil* (nothing). These are reserved words which you are not allowed to use as variable names.

The easy one to understand is *t*. For example, there is a function *equal* which compares two objects and tells you if they are equal. All of the following return *t*:

```
(equal 2 2)
(equal 10000 (+ 4000 6000))
(equal (doc–point–cmpn) (doc–point–cmpn))
```

```
(equal "howdy" "howdy")
(equal (+ 2 3) (+ 3 2))
```

But when two things are not equal, the *equal* function returns *nil.*

Nil in fact is an empty set, a list with nothing in it. That's why you'll sometimes see it expressed as *()*.

Nil is *not* zero. The expression *(– 5 5)* (which subtracts five from five) returns the number 0, not *nil.*

Variables

As with any programming language, with I-Lisp you can use variables. For example, suppose you wanted to remember what the line-spacing of a component was, perhaps so you could set it back to its original state after making some alterations. To do that, you have to tell the component to give you its line-spacing. Then you alter its line-spacing. Then you tell it to set its line-spacing back to the original. For this you need a variable to hold the value of the original line-spacing.

In some languages, if you want to set a variable called, say, *linespace* to the value 10000, you would use an expression such as *linespace = 10000.* With I-Lisp, you would say: *(setq linespace 10000)*. (You could also do this with the *let* command; we'll discuss the difference later.)

The following would also work:

```
(setq linespace (+ 6000 4000))
(setq linespace (doc–point–cmpn))
(setq linespace "Hi there")
(setq linespace (tell (doc–point–cmpn) mid:get–props :line–spacing))
```

Now, *linespace* is a pretty confusing name for a variable for storing a component object or the phrase "Hi there," but the point is that with I-Lisp, you don't have to say what *type* of variable a variable is. You can store any type of data in any variable, and you can change your mind and store a different type of data in it later if you'd like.

Using *setq* is also an I-Lisp instruction, of course, so it too returns something. It turns out that *setq* returns the new value of the variable. This is frequently useful in ways that you'll see demonstrated later.

In I-Lisp, the rules for what is a legitimate variable name are very loose. Variables can't consist only of digits (or else I-Lisp will think it's a number), but can be basically as long as you like — 50-character names are OK with I-Lisp if you really want to do all that typing, The convention is to separate the words in a variable by a dash. For example, typical variables might be: *number–of–right–answers, box–in–upper–left, hyper–jump–destination,* etc.

By the way, if you wanted to change the line spacing of a component and then change it back, here's a script that would do it (anything following a semicolon is a comment):

2–3 change–and–restore–line–spacing

```
; Tell the component to give its line spacing, and set original to that value
(setq original (tell (doc–point–cmpn) mid:get–props :line–spacing))
; Set the component's line spacing to 2000 rsu's
(tell (doc–point–cmpn) mid:set–props :line–spacing 2000)
; make sure the document is updated to reflect the change
(doc–flush–queue)
; Set the component's line spacing back to the original
(tell (doc–point–cmpn) mid:set–props :line–spacing original)
```

Lists

I-Lisp thinks in terms of lists. A list is just what you think it is — a set of objects one after another. I-Lisp doesn't care much about what sort of objects you put in a list even if they're all of different types. It even lets you include lists as objects in a list.*

A simple list might consist of some numbers *(1 2 3 5.6 6 7)*, of some strings *("one" "two" "three" "five point six" "seven")*, of some variables (doc–type cmpn–name first–word–on–page), or any other data. A list can mix various types *(1 2 "three" doc–type 5)* freely. It can even contain other lists *(1 2 (doc–type 4) (cmpn–name 5.6 "Howdy") 7)*. This last list consists of five items, two of which are themselves lists:

1. 1
2. 2
3. (doc–type 4)
4. (cmpn–name 5.6 "Howdy")
5. 7

A list may also contain an item such as *(+ 4000 6000)*. But I-Lisp will take that to be an instruction to be executed. So, the list (1 2 (+4000 6000) 4) would actually turn out to be the list (1 2 10000 4). Suppose for some reason you wanted to keep the expression (+ 4000 6000) on your list without having I-Lisp evaluate it and turn it into 10,000. You can do this by putting a single quote mark before it: *(list 1 '(+ 4000 6000))*. The quote tells I-Lisp not to evaluate what follows, but instead just to accept it.

---

*In this way, lists are unlike arrays in other languages. Both lists and arrays are data structures—ways of holding data—but you can get at any datum in an array by using its number, whereas to get at it in a list, you have to start at the beginning and count your way in. For example, suppose you wanted to find Jane Smith at graduation ceremonies. If there is an array of graduates and you know Smith is sitting in row 15, seat 12, you could find her by asking to see the person at "coordinates"15:12. On the other hand, if there is a list of graduates, especially if they're not in alphabetic order, you find Smith by beginning at the beginning and looking at every name until you find her. I-Lisp, based on Lisp, uses lists but provides shortcuts that give you array-like techniques for finding what you need in lists.

To create a list, I-Lisp provides a function called *list*.

Function 2–1: list

(list *object object, etc.*)

Returns: List of objects

E.g. *(list 1 2 (list "a" 3) 4 (+ 10 12))*
Returns: (1 2 ("a" 3) 4 22)

Here are some more examples:

(list 1 2 3)
Returns (1 2 3)
(list "one" 2 3)
Returns ("one" 2 3)
(list 1 2 3 (list 3 4 5))
Returns (1 2 3 (3 4 5))
(list 1 2 (+ 1 2))
Returns (1 2 3)
(list 1 2 '(+ 1 2))
Returns (1 2 (+ 1 2))

Lists can of course themselves be stored in variables or contain variables. For example: *(setq list–of–numbers (list 1 2 3))* would store in the variable *list–of–numbers* the list *(1 2 3)*. And those variables can be used in lists. For example, *(list list–of–numbers 4 5 6)* would return a list with four items, the first one of which is the list *(1 2 3)*.

List processing

You will learn more about list processing in Chapter 5, which is devoted to that topic. But you need to know a little now.

Every list can be thought of as having two parts: the first object and the rest — a head and all that follows. For reasons lost in the historical origins of Lisp, the head of a list is called the *car* and the tail is the *cdr* (pronounced "could-er").

Function 2–2: car

(car *list*)

Returns: First item on list

E.g. *(car (list (list 1 2) "a" "b"))*
Returns: (1 2)

Let's say you have the list *(1 "two" 3)* which you've *setq*'ed to the variable *my–list*. The instruction *(car my–list)* will return 1. Now you want to see the next item on the list, so you once again execute the instruction . It returns 1. This isn't what you wanted.

Where did you go wrong?

Easy. Asking to see the *car* of a list doesn't change the list. So every time you ask to see *the car*, the same old head shows itself.

Instead, you need to see the head and then behead the list. That's one place *cdr* comes in handy.

Function 2–3: cdr

(cdr *list*)

Returns: List of everything except the first item on list

E.g. *(cdr (list (list 1 2) "a" "b"))*
Returns: ("a""b")

If you want to look past the first element of a list, you can set a variable to be equal to the *cdr* of the list, and then look at that new *list's car.* For example:

```
(setq my-list (list 1 2 3))
Returns (1  2  3)
(setq rest-of-list (cdr my-list))
Returns (2  3)
(car rest-of-list)
Returns 2
```

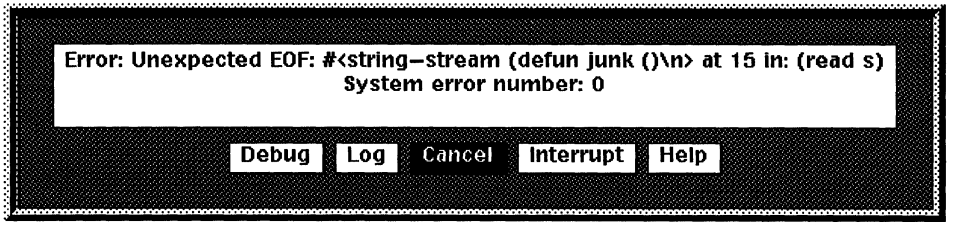
In later chapters you'll learn how to loop through a list to examine each of the members. You'll also learn some more direct ways of getting at members of lists.

Parentheses

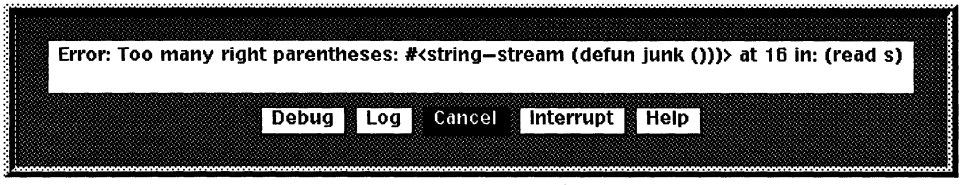
You have undoubtedly noticed the heavy use of parentheses in all of the examples. That's because parentheses are the walls of I-Lisp — they indicate where a function or a list begins and ends. You will certainly become familiar with the I-Lisp error message that tells you that you have unmatched parentheses. That may mean you made a typo, or it may indicate that you haven't correctly thought through the basic units or structure of your program.

The integrated programming editor that comes as part of the Developer's Toolkit provides parenthesis matching: put your text caret before a left parenthesis or after a right parenthesis, type ^Q, and your text caret will move to the matching parenthesis.

The parentheses tell I-Lisp about the structure of the expression. You need to pay attention to them. And you also need to use the tools that I-Lisp provides to help you get them right.



Short one parenthesis.



One too many parentheses.

Chapter 3
Working with Interleaf Lisp

In order to try out the Interleaf Lisp scripts you're going to learn how to write, you need to understand the mechanics of entering the text, loading the script, etc. That's what this chapter will teach you. (I will assume that you do not have the Developer's Toolkit option. If you do, you also have documentation explaining the tools that come in the Toolkit.)

In the course of this book, you'll be trying out lines of code and writing scripts. There are several ways of doing this.

- Lisp icon. Make a copy of the Lisp icon that comes with the system and open it to enter text.
- Word processor with a pure ASCII text mode. Use your favorite word processor, so long as you can save the document in pure text mode.
- Programming editor. Use a programming editor such as *emacs*.

We'll look briefly at each alternative.

Lisp icon

When you open a Lisp icon, you're given some choices. Tell it that you want to open it for edit. Now you can enter text into it. (If you do have the Developer's Toolkit, you'll have a very useful set of utilities available to you while you're within a Lisp icon).

The default Lisp icon that comes in the system's Create cabinet contains the following lines:

```
(lisp-set-implementation "Interleaf Lisp" "2.0")

;; Module name:
```

```
;; Purpose:
;; Notes:
;; Interfaces:

;; Audit:
;; DD-MON-YY USERNAME   COMMENT
```

Only the first line is really required (and the script will actually work even without it). The first line tells the system which version of Interleaf Lisp you're using. At some later date, if you don't have that line, the system may get confused because it's expecting Interleaf Lisp version 3.0 or 4.0. The rest of the lines in the default Lisp script are comments; the idea is that you fill in the blanks about what the name of the script is, what it's for, any random notes, which user interfaces it supports, and the date and creator's name. This information is completely optional, although it's handy for others who may come across your script.

To enter text, go past all of the comments and start.

Let's say you enter the line of text: *(stk-open "Hello")* and now you want to try it out. Here's how:

- Save the file by using the popup (or try ^X^S). Notice that you do not have to *close* the file.
- Select its icon on the desktop.
- Do a Custom->Load (using your desktop popup menu).

One of many easy mistakes to make is to alter what you've typed, select the icon, and load it, and notice that nothing different happens. That's because you forgot to *save* the file. When you *Load* a selected icon, it actually reads the file in from the disk; to alter the file on the disk, you have to save it.

There are advantages to working within a Lisp file:

- You get a set of useful tools for saving, searching and printing the file.
- You can work without leaving the Interleaf desktop.

There are also some disadvantages:

- The editing capabilities of the Lisp icon are not nearly as strong as in a professional programming editor.
- For very long files, a programming editor may be faster.

Word processor or programming editor

Instead of the Lisp icon, you can use either a word processor or a programming editor. The advantage of a programming editor is that it will have all sorts of useful tools (e.g., the ability

to show you where a parenthesis's mate is, the ability to automatically indent in ways that clarify the structure of progams), and probably will have better performance than a word processor. The disadvantage of a programming editor is that you probably don't own one, whereas you probably do have a word processor. On the other hand, there are a number of very good programming editors either in the public domain or free (e.g., many DOS editors available as shareware or freeware, and gnuemacs in UNIX).

To create an Interleaf Lisp script with either of these types of text editors, you create a file with a ".lsp" extension (e.g., "myscript.lsp"). (Try to keep your file names to eight letters or less, and don't use any peculiar characters; that will help make it easier to transfer your scripts from one type of computer to another.) When the Interleaf desktop sees the ".lsp" extension, it assumes that it is a Lisp script and will create the appropriate icon. (Remember to use Custom->Rescan if an icon doesn't show up when you think it should.)

To run a script you've created, save the file, select the icon, and do a Custom->Load, exactly as you would if typing into a Lisp icon. (Once again, remember to save the file in your word processor — but not necessarily close it — before doing a Custom->Load to ensure that the system is loading the current version of the script.)

The advantages of using either a word processor or a programming editor are that they offer a great deal of functionality and power. The disadvantage is that you have to work in a separate window, off your Interleaf desktop. For those running DOS, this means that you have to "shell" out of Interleaf, start up your text editor, make your changes, close out of your text editor, "exit" back into Interleaf, try out your changes, etc. For DOS users, working in the Interleaf Lisp icon probably makes more sense in most circumstances.

Recommendation

As a beginner, you may find that the Interleaf Lisp icon meets all your needs. If it does, it is slightly more convenient to stay within the Interleaf desktop.

As you begin developing longer and more complex scripts, if you are in a multitasking environment you ought to run a programming editor in a separate window. You get the functionality of the programming editor and the ability to try out your scripts quickly.

If you are not in a multitasking environment, use the Interleaf Lisp icon. If you start stretching its capacity, you can always break your script into shorter files and join them at the end.

Loading a script

Let's say you create a script that you actually find useful. There are a number of different ways to load (run) it.

- If it's a script you want to use in some sessions but not all, put it somewhere where you can find it. When you want to use it, do a Custom ▸Load.

- If it's a script you always want to use (e.g., it lets your ^O key open a template file you frequently use), put it in your *profile* drawer so that it will be automatically loaded every time you start up Interleaf 5. (If you use fast startup — you'll see the fast startup icon on your desktop, probably labeled "Ileaf5" — the changes in your *profile* drawer may be ignored. Purge the fast startup icon and start the system up again to cause it to reread the *profile* drawer's contents.)
- If it's a script that you want available from the desktop, put it in either your *Selection* or *No Selection* cabinet.
- You can attach a script to a keystroke so that whenever you press that key, the script runs. In fact, you can cause scripts to run on almost any user action, including selecting a graphic, leaving a window, and closing a document.
- By the time you read the end of this book, you'll be able to attach the script to a particular document or class of document so that its functionality will be available only in the documents you want it to be available in.

A first script

Now let's practice by writing and running some small scripts. (This section will repeat some of what you learned in the previous section.)

The first step is to choose one of the methods above to create a file that the system will recognize as an Interleaf Lisp script. Perhaps you will simply create a Lisp icon by using the system's Create menu. Or perhaps you'll create a file on the desktop with a ".lsp" at its end.

Now that you have a document that Interleaf 5 will recognize as an Lisp script, type the following lines into it. Watch those parentheses! (Any text from a semicolon to the end of a line is a comment and has no effect on the program itself; it is very valuable to comment as much of your code as possible so you can go back to it later on and understand what you did. Also, do not type in the boldfaced titles of the functions; they're not a part of the function itself.)

3–1 first–stickup

```
(stk–open "Hello, World")
```

Now save the file and run it by selecting it, and choosing *Load* from the *Custom* popup menu you get when you hold down your menu button while you're on the desktop. (Make sure the Lisp icon is selected or else you won't see *Load* on your *Custom* options.) Load the script. Lo and behold, a stickup opens.

Stk–open is a *function* that comes already predefined in Lisp. But suppose you want to add your own functions. Suppose, for example, you like your stickups to thank the user after they choose *Yes* on a yes–or–no stickup. As a first step, delete what you just typed and type in:

3–2 yes–or–no stickup

```
(stk–open "Answer yes or no" :yes–no)
```

In Lisp, every function (such as the one you just typed) returns some value after it is run. (In fact, you should think of these functions as being *evaluated* instead of run. Indeed, *eval* is the Lisp function that causes a script to "run.") In the case of a yes–no *stk-open*, the return is the value *t* (for a yes response) or *nil* (for a no response), depending on which button the user has pressed. You can use this return to cause another set of code to be *eval*'ed. The following will give a "Thank you" stickup if the user answers "yes" to the yes–or–no stickup:

3–3 polite–stickup

```
(if (stk–open "Answer yes or no" :yes–no)
  (stk–open "Thank you!"))
```

The first line creates the stickup and waits for the user to interact with it. The value *t* or *nil* is returned, depending on the user's choice. *If* the return of *(stk–open "Answer yes or no" :yes–no)* is *t*, then the next line of code is *eval*'ed, causing a stick-up to appear with the words "Thank you!".

Now, let's say this is such an incredibly useful capability that you would like to use it in many different programs, or at many different times within one program. Just as *stk–open* has a meaning within Lisp, you can give your polite stickup mini-program a name. You do this by defining it as a function, using the word *defun:*

3–4 polite–stickup–function

```
(defun polite–stickup ()
; Says thank you after answering yes to a stickup. Demo.
  (if (stk–open "Answer yes or no" :yes–no)
  (stk–open "Thank you!")))
```

Let's examine this function.

The first line says that we are creating a new function which will be named *polite–stickup*. (We'll explain the empty parentheses in a minute.) The next line explains what the function does. Because it begins with a semicolon, it is ignored when the function is *eval*'ed. The next two lines are the function itself. Notice that if you match the pairs of parentheses, there are none left over, and the very first one matches the very last one. (Having an editor that can show you where the matching parentheses are is very useful when working in Lisp.)

If you *eval* this new function, you'll see that no stickup shows up on your screen. That's because there's a difference between *defining* a function (which is what *defun* does) and *using* a function. Basically, you've created a new function that can be used just as the predefined ones can be. But how can you try it out?

With nothing selected on the desktop, do a Custom->ReadEval. This stickup allows you to enter a Lisp function and *eval* it, and returns the value in a stick up. So, if you were to enter (+ 4

5), the system would present a stickup giving you the result of the *eval*, which is 7.* Or type in *"(stk–open" Hi there")"* and press the Enter button, and it will *eval* that phrase and create a stickup with the text "Hi there." Just as you can use *ReadEval* to *eval* the "+" function or the "stk-open" functions, you can also *eval* the new function you've just defined. So, into the *ReadEval* stickup enter the phrase "(polite-stickup)" and click on the Enter button on the stickup. This will run the function and create the polite stickup.

While you are developing a script, you may not want to have to type the function name into the *ReadEval* stickup every time you want to run it. First of all, while the stickup is on your screen, ^P will cycle through the previous ten or so lines you typed into it, so you can go back to a previous entry without having to reenter it. Second, if you put the following line into a script in your Profile drawer, you will be able to get the *ReadEval* stickup by typing ^R while on the desktop:

(kbd–bind kbd–dt–map "\^R" '(ileaf:ileaf–listen :custom)).

Finally, probably the fastest way to run your script is to include at the bottom of the script itself the line that you would have typed into the *ReadEval* stickup. To stick with our polite stickup example, you would put the line *(polite–stickup)* at the end of the file you defined the function in. Then when you save your file, select the icon and Load it, it will see the *defun* and will make an updated version of the function you've defined, and it will see the *(polite–stickup)* and will *eval* it which will cause the function to run (exactly as if you had typed it into the *ReadEval* stickup). But remember that when you're done editing your script, you should remove the *(polite–stickup)* line unless you want it to run every time you load it. For example, if you were to put the script into your Profile drawer, every time you start up your system it will create the polite stickup and wait for you to answer yes or no.

Now that you've defined *polite–stickup*, you can use it just as you would any other Lisp function: *eval (polite–stickup)* and you'll get your stickup. (Because *polite–stickup* is one you defined and does not come as part of Lisp itself, next time you load Interleaf 5, it won't know what *polite–stickup* means; if you want to use your new function again, you have to *eval* its definition again by loading the script.)

Let's take one additional step here, just to give you a little more information about using *defun*.

Let's pretend you really care about politeness, so you want *all* the new stickups you create to be polite. And you don't want all your stickups to have the text "Answer yes or no." So, just as you can tell the standard Lisp function *stk–open* what text to use, you can tell a newly-created *polite–stickup–2* what text to use. You do this by passing *polite–stickup–2* an *argument*. Just as "Hello, World" is the argument in *(stk–open "Hello, World")*, "Shall I greet the world?" is the argument in *(polite–stickup "Shall I greet the world?")*.

---

*Just checking to see if you're paying attention.

But for *polite–stickup–2* to know what to do with an argument, we need to alter it:

3–5 polite–stickup–function–2

```
(defun polite–stickup–2 (text)
; Demo. Says thank you after answering yes to a stickup.
  (if (stk–open text :yes–no)
    (stk–open "Thank you!")))
```

Notice that we've renamed it, and we've put something between the parentheses on the first line. What's between the parentheses takes its contents from the argument passed to the function. So *text* now stands for the words "Shall I greet the world?" (but only within this function). In the third line, when the system sees the word *text*, it substitutes the argument passed to it.

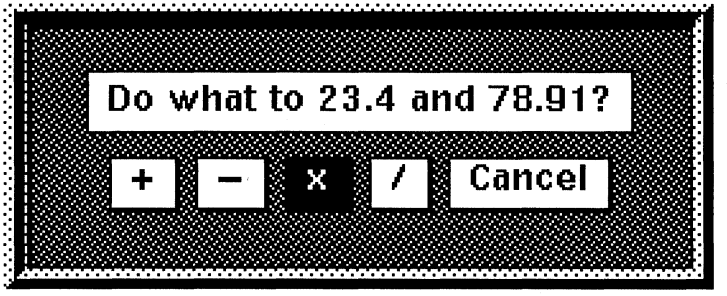
Functions can take more than one argument. Here's a function that takes two arguments. In this case, it takes two numbers and then gives you a stickup with special buttons for four different arithmetical operations. After the user chooses one, it puts the result of the calculation into a stickup. At this point, you will not be able to understand what's going in this function; it's really only here as an example of a function with two arguments. Sometime later you might want to look back at this and see how you could improve it to make it actually useful.

3–6 primitive–calculator

```
(defun primitive–calculator (number1 number2)
(let (operator answer) ; create two temporary variables called "operator" &
  "answer"
  ; make sure we're using floats, not integers
  (setq number1 (float number1))
  (setq number2 (float number2))
  ; create the stickup
  (setq operator
    (stk–open
      (format nil "Do what to ~D and ~D?" number1 number2)
      :buttons (list "+" "–" "x" "/" "Cancel")))
  ; do an operation, depending on which button was pressed
  (cond
    ; plus
    ((= 0 operator) (setq answer (+ number1 number2)))
    ; minus
    ((= 1 operator) (setq answer (– number1 number2)))
    ; times
    ((= 2 operator) (setq answer (* number1 number2)))
    ; divide
    ((= 3 operator) (setq answer (/ number1 number2)))
    (t (quit))        ; cancel
```

```
  )
  ; show the answer, converting it from a float to a string
  (stk-open (ftoa answer))
  ; return the answer
  answer
))
```

This function also returns the value of the answer, so if it is invoked by some other function, that other function could receive the answer and conceivably make some use of it.



Stickup asking for further input.

Attaching a script to a keystroke

Remember, there's a difference between defining a function and using it. Defining a new function is like putting a new tool into your toolkit. Using a function is like using the tool.

For example, suppose you need to do square roots sometimes. Here's a simple function that lets you type a number into a stickup and be shown in a stickup what its square root is.

3–7 square-root-stkup

```
(defun square-root-stkup ()
; Prompts for a number. Returns square root.
(let (n)
  (setq n (stk-open "Enter number" :input 10))
  (stk-open (ftoa (sqrt (atof n))))
))
```

Let's not worry how this function works. (In fact, the last line is sort of interesting because it converts a string to a number, gets the square root, and then converts the square root back to a string.) Let's just assume it works. Now you want to be able to use it while you're in a document. Let's make it so ^F (control–F) will invoke it. (This will also mean, however, that ^F will no longer move the text caret forward by one character, the way it usually does.) Go back to your Lisp document window and type in:

```
(kbd-bind kbd-doc-map "\^F" 'square-root-stkup)
```

This line says that from now on, when you're in a document, ^F will invoke the *square-root-stkup* function. (So will ^f; control characters don't care about capitalization.)

Save your Lisp file, select it, and *Load* it by using the *Load* command off the *Custom* entry on your desktop popup. Nothing visible happens.

While your mouse cursor is inside your Lisp window, type ^F . Still nothing happens. That's because you said ^F will invoke the square root function only when you're inside a normal document; that's what *kbd-doc-map* means. The Lisp document you're in is not a normal Interleaf 5 document.

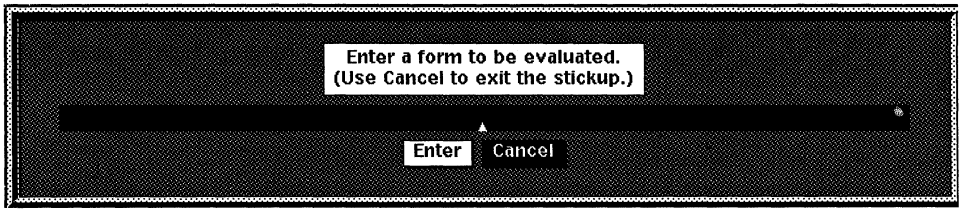
But now move your mouse cursor inside a normal Interleaf document window. Type ^F. And, sure enough, a stickup appears. (If it doesn't say "Enter number" — if it contains some information about Lisp errors — check your typing and then reread this chapter to this point.)

You have created a Lisp script that creates a new function and that says that whenever you type ^F (when you're in a document) that function ought to be invoked.

In general, that's how Lisp works. You create potential actions and specify the actions or conditions that will cause the system to take those actions.

Breaks

There is a function that is extremely helpful while developing a program: *(break)*. You stick in a breakpoint where you want to be able to stop and look around at what's happening. When *break* is *eval*'ed, it puts up a stickup that lets you check the value of variables and get other information (the same stickup you get if you run *ReadEval* off your desktop Custom nothing-selected popup). (If you have the Developer's Toolkit, it puts you into a "listener" which makes it even easier to get information.)



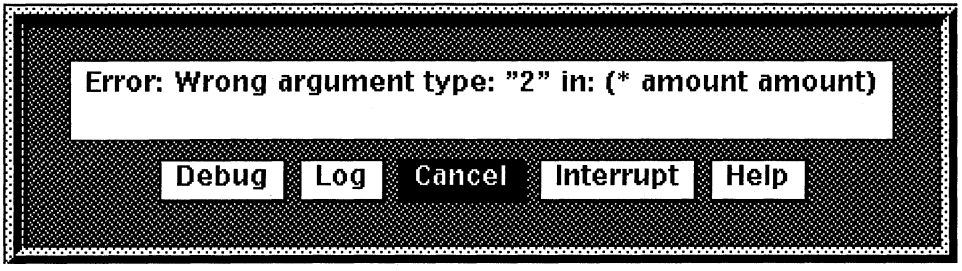
ReadEval stickup.

For example, here's a function that doesn't work:

```
(defun square-junk ()
; squares any number entered
```

```
(let (amount total)
  ; get an amount to square
  (setq amount (stk-open "Enter amount to square" :input 10))
  ; square it
  (setq total (* amount amount))
  ; return it
  total
))
```

When you try to run this, after you enter some amount into the stickup (say, 2), you'll get the following error message:



Error stickup.

If you choose "Debug," you'll get the *ReadEval* stickup. Into it you can type any variable or function and it will *eval* it and return the result to you in another stickup. So, if you type "amount," it will return "2" to you. (And if you pay careful attention, you'll notice that since the numeral is in quotes, *amount* is a string, not a number and thus it cannot be multiplied and thus you get the error message.)

You can continue running your program from where the break occurred by typing *(continue)* into the stickup.

(Note that ^Y and ^P will cycle through previous entries you've typed into the stickup.)

As you develop programs, you'll find yourself taking *break*s frequently.

Chapter 4
Basics of Programs

In this chapter we'll look at the basic elements of programs — how to assign values, test values, and build control structures, and how to build a useable function.

Setting values

I-Lisp provides several flexible ways of setting a symbol equal to a value. Perhaps the one with the most general utility is *setq*.

Function 4–1: setq

(setq *symbol value*)

***Returns*: the value**

E.g. *(setq age 13)*
Returns 13 and age *is now set to 13*

Basically, *(setq x y)* is I-Lisp's way of saying *x* now equals *y*.

Setq also lets you assign values to a bunch of symbols all at once. For example:

```
(setq    one     1
         two     2
         three   3
         rest    (list 4 5 6))
```
Returns (4 5 6)

This line of code has set *one* equal to 1, *two* equal to 2, *three* equal to 3, and *rest* equal to the list (4 5 6). It's just a shortcut.

Because *setq* returns the value of the last symbol it sets, you can write very compressed code. For example:

```
(+ (setq age 3) 4)
```
Sets "age" to 3 and returns 7

In I-Lisp, as you'll eventually see, mathematical expressions put the math symbol first, rather than between the two items being affected; *(– 4 3)* is I-Lisp's way of saying *(4 – 3)*. Now let's consider the line of code above. This line is evaluated, as always, from the inside out. At the inside is the expression *(setq age 3)* which makes *age* equal to 3. But it also returns 3. So *(setq age 3)* gets treated as its return and then gets added to 4. So, the line of code above not only adds 3 and 4, but sets *age* equal to 3.

This becomes especially important in control loops, as we will discuss below. Here's a quick preliminary example:

```
(setq stock (list 145 645 243))     ; make a list of 3 items in stock
(while (setq item (pop stock))      ; look at each item in list
  (do-something-with item))         ; if there is one, go do something with it
```

In this example, we first create a list (*stock*) using *setq*. Then we use the *while* statement we'll describe below in detail. Basically, *while* looks at an expression (in this case, *(setq item (pop stock))*) and, if it isn't nil, does what follows. (Presumably somewhere we have defined a function called *do–something–with*; we won't worry about that here.) Then it goes back, checks the initial expression again, and loops until the expression returns nil.

In our example, the *while* statement checks whether *(setq item (pop stock))* returns nil. Now, *(pop stock)* returns the first item on the list *stock* and removes that item from *stock*. If there are no items left in *stock*, it returns nil. Everytime *stock* is *pop*ped, *item* is *setq*'ed to the first element of *stock*. When *stock* is empty, the *pop* statement returns nil. That means *item* is set to nil. And that means the *setq* statement returns nil. And that, in turn, stops the *while* statement. So, those few lines of code look at every item in turn and stop when there are no more items left.

This is a very common sort of construct; you will see it frequently in examples on your Interleaf system. If you are confused by the example, come back to it after you have read the section on *while*.

By the way, if you try to do some things with a symbol before you have introduced it to the system by *setq*ing it, you will get an I-Lisp error. For example, if you use *push* (to which you will be formally introduced later) to put an item on to a list, you have to make sure the variable representing the list already exists:

```
; wrong way
(push 1 age-list)
Returns error message: "Symbol not bound: age-list"
; right way
(setq age-list nil)
(push 1 age-list)
Returns (1)
```

Function 4–2: let

(let *(symbols) code)*

***Returns*:** nil

Let is used primarily to establish symbols for use only for the duration of a function. For example:

```
(let (x)
   (setq x 20)
   ; x is now set to 20
) ; end the scope of let
   ; x now isn't set to anything; using it returns an error message
```

The scope of the *let* is everything between the parenthesis that prefaces it and the matching parenthesis. Immediately after the *let* comes a list of symbols, all within parentheses. Within the parentheses containing the list of symbols can be symbols set to values; these are themselves within parentheses. A commented example will help:

```
(let
   (x                          ; begin list of symbols and create x
   (month "July")              ; create month and set it to "July"
   (days (list "Sun" "Mon"))   ; create days and set it to Sun, Mon
   (count (+ 4 6))             ; create count and set it to to 4 + 6
   )                           ; end list of symbols
   (car days)                  ; returns "Sun"
   (push 2 time)               ; returns error — time's not a symbol
)                              ; end of scope of let
```

Why would you want to use *let*? When you start putting together complex programs, especially ones that use functions you may have written a long time ago, or functions you've borrowed from other people, or functions that use established libraries of functions, you can't always be sure that some other function hasn't already used a symbol with the same name as one you've created. For example, suppose you create a symbol called *counter* which in a particular function is used to count the number of items in a list. It gets set to, say, 45. Suppose some other function — perhaps used in a completely different program later in the day — also uses a symbol called *counter*. You don't want your value of *counter* (45) to replace the one the other program expects.

So Interleaf Lisp lets you define symbols that are used only during a *let* statement, typically within a single function. As soon as that function is over, it's as if you never defined the variable at all.

So, a typical function might be structured as follows:

```
4–1 select-page
(defun select–page ()
; selects all the cmpns on a page
```

```
(let (page  first-c c)
 ; get current cmpn
 (setq first-c (doc-point-cmpn))
 ; deselect all cmpns
 (tell *cmpn-editor* mid:deselect :all)
 ; select current cmpn
 (tell *cmpn-editor* mid:select first-c)
 ; get current page
 (setq page (doc-page-of first-c))
 ; walk up, selecting all cmpns on that page
 (setq c first-c)
 (while (and (setq c (tell c mid:get-previous :along :structure))
   (eql page (doc-page-of c)))
   (tell *cmpn-editor* mid:select c))
 ; walk down
 (setq c first-c)
 (while (and (setq c (tell c mid:get-next :along :structure))
   (eql page (doc-page-of c)))
 ; make sure the document is up to date
(doc-flush-queue)
))
```

Now you could attach this to a keystroke:

```
; attach this to control-X p
(kbd-bind kbd-doc-map "\^Xp" 'select-page)
```

If you try to use *page* or *first-c* outside of *select-page*, you'll get an error message saying that the symbol isn't bound.

Sometimes, however, you want a symbol to be recognized everywhere, not just in a function. These are called *globals*, and they must be used with care because if someone else has created a global with the same name, you are likely to trash each other's programs. The convention in Interleaf Lisp is that globals begin and end with asterisks, e.g., **month**, **list-of-restaurants**, etc. (Sure, the asterisks make globals harder to type — that's probably why it's the convention.)

Interleaf provides a very solid mechanism for ensuring that symbol names never conflict: packages. Developers can establish a package for their program (or suite of programs) which causes Interleaf internally to recognize that all symbol names in those programs are to be used only within those programs. So, if two packages both use the symbol *x*, Interleaf will keep them distinct. Unfortunately, the use of packages is beyond the scope of this book. As an alternative, you are strongly advised to use unique names.

One way of doing this is to make sure that all your functions are prefaced by a two- or three-letter prefix unique to the program you're creating. For example, if you're writing one

called "Baseball Scores," all the functions you use could be prefaced by *bbs–*, e.g., *bbs–get–data, bbs–count–strikes, bbs–rbi.*

Then, within functions, use *let* so that you don't have to worry about symbols clashing — the symbols you create within *let* won't be recognized outside of the *let.**

Function 4–3: defvar

(defvar *variable value*)

***Returns*: value**

E.g. *(defvar minimum–age 17)*
Returns minimum–age as a symbol

Defvar is used to create global variables. If the variable you're trying to set is already a global variable (e.g., you've *setq*'ed it earlier, not within a *let*), then the variable retains its original value.

Notice that because *defvar* returns the symbol, and not its value, the following will not work:

(+ (defvar x 3) 4)
Returns Lisp error

To I-Lisp, this looks like you're trying to add 4 and a symbol (*x*). After you've set *x* to 3 using *defvar*, however, you can use *x* interchangeably with 3. So, the following does work:

(defvar x 3)
Returns x
(+ x 4)
Returns 7

Function 4–4: boundp

(boundp *symbol*)

Returns*: *t* if *symbol* is bound; otherwise, *nil

E.g. *(setq x 1)*
(boundp 'x)
Returns t

A symbol is *bound* if somewhere along the line it's been given a value, even if that value is nil. If it's been given a value within a *let* statement, it's only bound within that statement. Sometimes you need to find out if a symbol has already been bound, either to prevent an error in a routine that assumes that it has been bound, or to help debug a program. *Boundp* will tell you if it's bound. (Notice the initial quote before the variable, telling the system not to evaluate the symbol, but simply to look at it as a symbol).

Boundp is also useful to see if you've already run a program. For instance, you can in a program set a global variable to *t*. The next time you run the program, if there are parts that

*There is one useful exception to this. Interleaf Lisp uses dynamic scoping. That means that symbols declared within a function through *let* will be passed to any other function called within that first function.

are only required to be run the first time the program runs, you can check to see if the global is bound. If it is, then you can skip the unnecessary pieces of code. For example:

```
(if (not (boundp '*my-font*))
  (setq *my-font* (list "Swiss" 16.5 :underline t)))
```

The first time this code gets *eval'*ed, **my-font** hasn't yet been bound to anything, so the *if* statement is true (i.e., it's *t* that **my-font** isn't yet bound). So the next line gets *eval'*ed, binding **my-font** to a list of font-like properties. The next time, however, that this code gets *eval'ed*, **my-font** will be bound, so the *if* statement will be *nil* and the rest of the statement won't be *eval'*ed. Not only does this make for marginally better performance, there are times when you definitely want some code to be executed once and only once. For example, the code that creates a new class of document will create multiple classes with the same name if you run it multiple times, so you'll definitely want to make sure that you run that code only once.

Function 4–5: makunbound

(makunbound *symbol*)

Returns*: *symbol

E.g. *(setq name "Jones")*
Returns "Jones"
(makunbound 'name)
Returns "Jones" and sets name *to* nil

Makunbound makes a symbol unbound. This is especially useful when debugging in order to return a program to its original state.

Function 4–6: fboundp

(fboundp *symbol*)

Returns*: *t* if *symbol* is bound; otherwise, *nil

E.g.
```
(defun test-fcn ()
        (do-something))
(fboundp 'test-fcn)
Returns t
```

This does the same thing as *boundp* but it checks whether a function, not a variable, has been bound.

Function 4–7: fmakunbound

(fmakunbound *symbol*)

Returns*: *symbol

E.g.
```
(defun test-fcn ()
        (do-something))
(fmakunbound 'test-fcn)
Returns "Jones")
```

This the same as *makunbound* but it unbinds functions rather than variables.

Testing values

Now that we have some variables loaded with values, we may want to test whether they're equivalent or not.

There are various tests of equivalency in Interleaf Lisp. Some only work with particular types of data, e.g., number or strings. Others are more forgiving. We'll look at these tests in rough order of flexibility, the narrowest tests first.

Function 4–8: =

(= *number1 number2*)

***Returns*: t if *number1* equals *number2*; else,** nil

E.g. *(= 4 (+ 1 3))*
Returns t

The equal sign tests two numbers. If you use it to test, say, two strings (e.g., *(= "Yes" "Yes")*, you'll get an error message.

Function 4–9: string=

(string= *string1 string2)*

***Returns*: t if *string1* is the same as *string2*; else, nil**

E.g. *(string= "Yes" "Yes")*
Returns t

String= will give an error message unless the two things it's comparing are both strings. For the strings to be equal, their capitalization must be the same (i.e., the test is case-sensitive).

Function 4–10: eql

(eql *object1 object2)*

***Returns*: t if *object1* and *object2* are the same; else,** nil

E.g. *(eql 5 5)*
Returns t

Eql works for a wide variety of objects, including document objects such as components and frames. There's really only one type of object *eql* won't work on: objects that consist of sets of other objects. For example, you can't use *eql* to compare two strings because strings consist of sets of characters. You can use *eql* to compare two lists, however.

If you pay attention to the following example, you will avoid a common and hard-to-debug mistake:

```
(setq list1 (list 1 2))      ; create a list
(setq list2 list1)           ; set another variable equal to the first
(setq list3 (list 1 2))      ; create a new list with same content as the first
(eql list1 list2)            ; test first two with eql
Returns t
(eql list1 list3)            ; test first and third with eql
```

Returns nil

In this example, we have set *list2* to be one and the same thing as *list1*. You can prove this by using *selt* (explained later) to alter one of the lists:

```
(selt 3 list2 1)        ; replace 2nd element of list2 with the number 3
Returns 3
list2                   ; look at content of list2
Returns (1 3)
list1                   ; look at content of list1
Returns (1 3)
```

Changing the content of *list2* also changes the content of *list1* also because you have told *list2* to point to exactly the same entity as *list1*. But *list3* is different: it's a copy of *list1* which has the same contents without being precisely the same thing (which is what being a copy means, after all). Because *list1* and *list3* are different lists, *eql* reports they are different, for they are different objects. (This makes sense. For example, two paragraphs may have the same contents but still be two separate paragraphs.)

Also, remember that integers are a different type of number than a real number. For example:

```
(eql 100 100.0)
Returns nil
```

The decimal point after the second *100* indicates that it is a real number that happens not to have any numbers to the right of the decimal point. But because it is a different type of number from the first *100*, *eql* reports that they are not identical. (The function *(= 100 100.0)* would return *t*.)

Function 4–11: equal

(equal object1 object2)

***Returns*: t if *object1* and *object2* have the same content; else,** nil

E.g.
```
(setq a (list 1 2))
(setq b (list 1 2))
(equal a b)
Returns t
```

Equal will test any two objects, even compound objects such as strings. Further, it doesn't demand that the two objects be one and the same, only that they have the same contents. So, using the example in the section above (on *eql*), *(equal list1 list3)* would return *t*.

Equal (like *eql*) does not consider an integer to be equal to a real number with the same value.

Function 4–12: equalp

(equalp object1 object2)

***Returns*: t if *object1* and *object2* have the same content; else,** nil

E.g. *(setq a (list 1 2))*
(setq b (list 1 2))
(equal a b)
Returns t

The major difference between *equal* and *equalp* is that *equalp* counts as equal two strings that have different capitalization. For example:

(equalp "Polish" "polish")
Returns t

In addition, *equalp* will accept as equal an integer and a real number with the same value. For example:

(equalp 100 100.0)
Returns t

Logical operators

Through logical operators (*and, or, not*) you can evaluate sets of statements to see if they are true.

Function 4–13: and
(and *statement1 statement2 ...*)
***Returns*: value of last statement evaluated**

E.g. *(setq x 3)*
(setq list1 (list 1 2))
(setq list2 list1)
(and (eql list1 list2) (eql 3 x))
Returns t

And looks at each of the following statements, and evaluates them until it comes upon one that returns nil. If it finds one that returns nil, then the *and* statement itself returns nil. If it doesn't find any nil*s*, then it evaluates all the statements and returns the value of the last one it evaluated.

Notice that once it has found a single statement that is nil, it stops evaluating the rest in the series. This can have serious consequences if you are counting on setting some value in the *and* statement itself. For example:

```
(and
  (eql x y)
  (setq z y))
```

If *x* and *y* are not *eql*, then the line *(setq z y)* will never be reached, so *z* won't be *setq*'ed to *y*.

In logic class you may have learned that an *and* statement is true only if all of its conjuncts are true. In Interleaf Lisp class you have just learned that an *and* statement evaluates to *non*-nil if all of its conjuncts evaluate to *non*-nil; in I-Lisp, the conjuncts can evaluate to anything except nil for them to satisfy an *and* statement. For example:

(setq x 1)
Returns 1
(setq y 2)
Returns 2
(setq z 3)
Returns 3
(and x y z)
Returns 3

This returns 3 because the last statement it evaluated was *z* and *z*'s value is 3.

And (and the other logical operators) is of use primarily in conjunction with control structures such as *if* and *while*. As you will soon see, those control structures go forward so long as the condition they're evaluating doesn't work out to nil. In English, we might put an *if* statement as "If this isn't nil, then do that." The *this* can be a number, a word, a function, or a component to satisfy the criterion. In short, in Interleaf Lisp, *not*-nil is often more important than *t*.

Function 4–14: or

(or *statement1 statement2 ...*)

***Returns*: value of first non-nil statement**

E.g. *(setq x 1)*
(setq y 2)
(or (eql x y) (eql y y))
Returns 2

Unlike *and*, *or* only needs one *non*-nil statement to return a *non*-nil value.

Whereas *and* stops evaluating statements once it's found one that is nil, *or* stops evaluating them once it's found one that isn't nil.

Function 4–15: not

(not *statement*)

***Returns*: t if *statement* is nil; *else* returns nil**

E.g. *(setq x 1)*
(not (eql x 1))
Returns nil

Not takes a non–nil value and returns nil, or takes a nil value and returns t.

You can use *not* not only to evaluate conditions, but also to flip a value. For example:

(setq heads t)
(setq tails (not heads))
Returns nil; tails is set to nil but heads remains t
(setq heads (not heads))
Returns nil; heads is set to nil

Chapter 5
List Processing

Lisp likes lists. In fact, its name derives from *list processing*. While there's plenty you can do in I-Lisp without ever encountering a list, sooner or later you're going to have to understand how the world looks to a list processing system.

This chapter will show you the basics of working with lists.

What is a list?

A list is a set of items arranged one after another. In I-Lisp, lists can contain any jumble of data types you want, including other lists. So, the following are acceptable lists (with parentheses marking the beginning and end of lists):

(1 2 3 4 5)
(3 2 4 1 5)
(1 "two" 3 4 5)
(1 2 3 (4 5) 6)
((Jim (34 "Male" "NY")) (Mary (35 "Female" "NJ")))

This last example actually is a list that contains two other lists, each of which in turn contains one list. The two items on the list are (Jim (34 "Male" "NY")) and (Mary (35 "Female" "NJ")), and the second item on each of these lists is itself a list containing information about age, sex, and location.

The ability of I-Lisp lists to be so freely structured and to contain such a wide variety of data is very important. About all that lists have in common is that the data in them is stored in sequence, one item after another.

Lists turn out to be a very useful way of looking at the world. For example, a document can be viewed as several different types of lists: a list of components, of words, of pages, of chapters. Taken as a whole, a document can be viewed as a list of lists. For example, this

chapter consists of a list of subsections, each of which contains a list of paragraphs, each of which contains a list of words and pictures, each of which contains a list of letters or picture elements.

As you learn Interleaf Lisp, you'll be surprised at how strong a model the list model is.

Making lists

There are several ways of constructing a list.

Function 5–1: list

(list *items*)

***Returns*: the list**

E.g. *(list 1 2 3 (list 4 5))*
Returns (1 2 3 (4 5))

The command *list* takes all the entries after it and turns them into a list. So, *(setq week (list "Monday" "Tuesday" "Wednesday" "Thursday" "Friday" "Saturday" "Sunday"))* would create a variable *week* and set it equal to a list of the names of the days.

There's a shortcut for constructing a list: put a single quotation mark outside a parentheses. So *'(1 2 3)* is the same as *(list 1 2 3)*, and *'(1 2 3 '(4 5))* is the same as *(list 1 2 3 (list 4 5))*

As always you can have I-Lisp do some extra work for you while constructing a list. For example, (+ 3 5) is the I-Lisp code that causes 3 and 5 to be added. If you place that code as an entry on a list, it gets replaced by the sum of 3 and 5. E.g., *(list 1 2 (+ 3 5) 4)* returns *(1 2 8 4)*. If you want the phrase (+ 3 5) to be on the list without being evaluated, put a single quote mark in front of it, e.g. *(list 1 2 '(+ 3 5) 4)* returns *(1 2 (+ 3 5) 4)*.

Function 5–2: append

(append *list item*)

Returns*: a new list with *item* as the last member of *list

E.g. *(setq flavors (list "vanilla" "chocolate"))*
(setq flavors (append flavors "strawberry"))
Returns ("vanilla" "chocolate" . "strawberry")

In the example, we first construct a list named "flavors" with contents of "vanilla" and "chocolate." We then want to add another flavor. *Append*ing "strawberry" returns a *new* list with "strawberry" at the end. Because it's a new list, *flavors* hasn't been affected. That's why you have to *setq flavors* to the new list. (Don't worry about the dot between "chocolate" and "strawberry" in the final line of the example above. It's not a typo, but it's also not real helpful to know about yet.)

Append is a relatively slow function (because it *eval*s every item in the list in order to (reach the end). Try using *push* instead, when possible:

Function 5–3: push

(push *item list*)

***Returns*: the list with the new item as its first element**

E.g. *(setq flavors (list "vanilla" "chocolate"))*
(push flavors "strawberry")
Returns ("strawberry" "vanilla" "chocolate")

Push varies from *append* in two ways. First, the item you *push* onto a list becomes the first element of the list, not the last. Second, push alters the list itself; *append* creates a new list. For example:

(setq flavors (list "vanilla" "chocolate"))
flavors now equals the list ("vanilla" "chocolate")
(append flavors "strawberry")
flavors still equals ("vanilla" "chocolate")
(setq flavors (append flavors "strawberry"))
flavors now equals ("vanilla" "chocolate" "strawberry")
(push "pistachio" flavors)
flavors now equals ("pistachio" "vanilla" "chocolate" "strawberry")

You cannot *push* an item on to a nonexistent list. Make sure you create the list first, even if it's an empty list, e.g., *(setq flavors nil)*.

Function 5–4: cons

(cons *item item*)

***Returns*: a two–item list**

E.g. *(cons "NY" "Albany")*
Returns: ("NY" . "Albany")

A *cons* is a special type of list. It consists of two items. (Of course, the items may themselves be complex lists.) A cons is expressed as a *dotted pair:* the two items are shown with a period between them.

There are special techniques for retrieving items from a list of conses which make conses especially useful. (See *assoc* and *rassoc*.)

Here are some examples of conses and lists of conses.

(cons 1 2)
Returns (1 . 2)
(cons 1 (list 2 3))
Returns (1 . (2 3))
(cons 1 (cons 2 3))
Returns (1 . (2 . 3))
(cons (cons 1 2) (cons 3 4))
Returns ((1 . 2) . (3 . 4))

(list (cons 1 2) (cons 3 4) 5 6)
Returns ((1 . 2) (3 . 4) 5 6)

Finding items in lists

The point of using a list is not to build an interesting list for its own sake but to be able to get information out of it. I-Lisp has a rich set of functions for doing so.

Those functions begin with the mysteriously named *car* and *cdr* (pronounced *could-er*), familiar to any Lisp programmer.

Function 5–5: car

(car *list*)

***Returns*: first element of list**

E.g. *(setq n (list 1 2 3 4))*
(car n)
Returns 1

Function 5–6: cdr

(cdr *list*)

***Returns*: the list minus the first element**

E.g. *(setq n (list 1 2 3 4)*
(cdr n)
Returns (2 3 4)

Car and *cdr* are surprisingly powerful functions. You can use them to build sophisticated functions for processing lists. (On the other hand, I-Lisp already contains many of those functions.)

Here are some more examples:

(setq precipitation (list "clear" "snow" "rain" "hail" "frogs"))
Returns ("clear" "snow" "rain" "hail" "frogs")
(car precipitation)
Returns "clear"
(cdr precipitation)
Returns ("snow" "rain" "hail" "frogs")
(car (cdr precipitation))
Returns "snow" [the first element of the cdr of precipitation]
(setq weather (list (list "clear" "snow" "rain" "hail" "frogs") (list "hot""cold""moderate")))
Returns (("clear" "snow" "rain" "hail" "frogs") ("hot" "cold" "moderate"))
(car weather)
Returns ("clear" "snow" "rain" "hail" "frogs")
(cdr weather)
Returns (("hot" "cold" "moderate"))

```
(car (cdr weather))
Returns ("hot" "cold" "moderate") [Notice single parentheses]
(car (car (cdr weather)))
Returns "hot"
(cdr (car weather))
Returns ("snow" "rain" "hail" "frogs")
(car (car  weather))
Returns "clear"
```

As the examples show, you can ask for the *cdr* of a *car* of a *car* of a *cdr*, if you want. I-Lisp looks at such a statement, begins with the innermost function, and works its way outwards. For example:

```
(setq n (list 0 (list (list 1 2 3 4) 5 6 7) 8 9 10))
Returns (0 ((1 2 3 4) 5 6 7) 8 9 10)
(cdr (car (car (cdr n))))
Returns (2 3 4)
```

Let's take this step by step, beginning with the innermost (in this case, rightmost) *cdr.* The *cdr* of *n* is (((1 2 3 4) 5 6 7) 8 9 10) — that's what you get if you take the first element (the 0) off. The *car* of that *cdr* is ((1 2 3 4) 5 6 7). The *car* of that in turn is (1 2 3 4). And the *cdr* of that is (2 3 4). (Keep re-reading this until it makes sense, and then take two aspirin and start again in the morning. The key is understanding the structure of the list; after that, *cdr* and *car* are easy.)

Notice that *cdr* and *car* do not modify the lists themselves. They report to you what the first element is and what the rest of the list is, but they do not change the list. So, if you are trying to examine the contents of a list, repeatedly applying *car* will keep giving you the same first element:

```
(setq n (list 1 2 3 4))
Returns (1 2 3 4)
(car n)
Returns 1
(car n)
Returns 1
```

How, then, do you look at each member of a list in succession? If you insist on using *car* and *cdr*, you could do use the *while* structure which we won't talk about until the next chapter. (*While* loops until the condition following it is nil.)

```
(setq n (list 1 2 3 4))
(while n                          ; loop until n is empty
  (setq item (car n))             ; get the first item
  (setq n (cdr n))                ; make n equal to its cdr
  (do-some-function item)))       ; do some function with item
```

There is, however, an easier way to accomplish the same result: *pop*.

Function 5–7: pop

(pop *list*)

***Returns*: first item on list**

E.g. *(setq n (list 1 2 3))*
(pop n)
Returns 1 and changes n to (2 3)

Unlike *cdr* and *car*, *pop* not only gets you the first item, it actually affects the list. It beheads the list. So, you could replace the code given in the previous section with:

```
(setq n (1 2 3 4))
(while n                        ; loop until n is empty
  (setq item (pop n))           ; get the first item and make n equal to its cdr
  (do-some-function n))         ; do some function with n
```

In fact, you could even save some space by writing this function as:

```
(setq n (1 2 3 4))
(while (setq item (pop n))
  (do-some-function n)) ; do some function with n
```

(This works because *setq* returns the value you are *setq*'ing to.)

Pop is frequently used in conjunction with *push*. *Pushing* an item on to a list places it at its front. As you repeatedly push items on to it, you're building a list in which the first elements come last and the last ones come first. *Pop*ping elements takes them from the front, so the most recently *push*ed elements come out first. For example:

```
(setq n nil)                    ; you can't push onto a non-existent list
Returns nil
(push 1 n)
Returns (1)
(push 2 n)
Returns (2 1)
(pop n)
Returns 2                       ; n is now equal to (1)
(pop n)
Returns 1                       ; n is now equal to (), or nil
(pop n)
Returns nil                     ; popping an empty list returns nil
```

If, for some reason, you are unhappy with having a reverse order list, you can use *reverse* to change the order.

Function 5–8: reverse

(reverse *list*)

***Returns*: the list in reverse order**

E.g. *(setq n (list 1 2 3 4))*
(setq back–n (reverse n))
Returns (4 3 2 1)

Reverse only reverses the top level of a list. For example:

(setq capitals (list (list "UK""London") (list"USA" "Washington")))
Returns (("UK" "London") ("USA" "Washington"))
(setq capitals (reverse capitals))
Returns (("USA" "Washington") ("UK" "London"))

Reverse does not actually reverse the list itself; it only returns a copy of the list in reverse order. So, in the example above, the original list (*capitals*) retains its original order.

Function 5–9: nth

(nth *number list*)

Returns*: item on *list* at position *number

E.g. *(setq n (list "a" "b" "c" "d"))*
(nth 0 n)
Returns "a"

Nth assumes the first element is numbered 0, so *(nth 3 test–list)* will return the fourth element of *test–list.*

For the first ten elements of a list, there is a slight shortcut: *(second test–list)* is the same as *(nth 3 test–list)*, *(third test–list)* is the same as *(nth 4 test–list)*, etc. And *(first test–list)* and *(last test–list)* also perform as expected.

Function 5–10: elt

(elt *list number*)

Returns*: item on *list* at position *number

E.g. *(setq n (list "a" "b" "c" "d))*
(elt n 0)
Returns "a"

This looks a lot like *nth*, except the order of *list* and *number* is reversed. *Elt*, like *nth*, is zero-based.

Elt also lets you count backwards, from the end of the list. It assumes that the last item on the list is –1, the second to last is –2, etc. For example:

(setq n (list "a" "b" "c" "d"))
Returns ("a" "b" "c" "d")
(elt n 1)
Returns "b"

(elt n −3)
Returns "b"

Function 5–11: member

(member *item list &optional test)*

Returns: list from matched item to end, or nil if no match

E.g. *(setq n (1 2 3 4 5))*
(member 3 n)
Returns (3 4 5)
(member 6 n)
Returns nil

Member lets you look for an element in a list. It doesn't just tell you whether the element is there or not; it actually tells you what the rest of the list is if the element is found on it.

Member is one of the I-Lisp functions that allows you to specify what test you want to use when you are trying to match an item on the list. If you don't specify any, it will use *eql*. You might prefer the more forgiving *equal*. For example, two identical strings will fail the *eql* test, but will pass the *equal* test. For example:

```
(setq n (list "a" "b" "c" "d"))
Returns ("a" "b" "c" "d")
(member "b" n)                    ; defaults to using eql as the test
Returns nil
(member "b" n 'equal)
Returns ("b" "c" "d")
```

You can even substitute your own functions instead of *eql* or *equal*. For example, we'll create a function called "last-name" which will find the last name in a two-word string and compare it with the last name in another two-word string.

```
(setq names (list "Mary Jones" "Phil Philby" "Ann Gifford"))
Returns ("Mary Jones" "Phil Philby" "Ann Gifford" )
(defun last−name (a b)
(let (l−name−a l−name−b)
   ; Get last names of a & b by looking for first space in each name
   (setq l−name−a (substring a (string−contained " " a)))
   (setq l−name−b (substring b (string−contained " " b)))
   ;are the two last names the same?
   (string= l−name−a l−name−b)
))
(member "Phil Nemo" names 'equal)
Returns nil
(member "Phil Nemo" names 'last−name)
Returns nil because "Nemo" isn't a last name on the list
(member "Annabel Philby" names 'last−name)
Returns ("Phil Philby" "Ann Gifford")
```

It's probably worth explaining this. We create a function called *last–name* that expects two arguments (*a* and *b*). The *member* function looks at every member of the list, one by one. For each member, it goes to *last–name*, sending it as arguments the item you're checking for inclusion in the list and the list item currently being checked. *Last–name* does whatever it is that you want, and sends the result back to *member*. In this example, *last–name* gets the last name of the name we're checking and the name on the list being checked, and reports *t* if the two are the same. *Member* stops checking the list items if and when one of the items on the list passes the test (i.e., in this case, the last names match).

One of the many uses of *member* is to help build a list that has no repeated members. For example, if you want to build a list of all the months that have a birthday of one of your co-workers, you could do something like the following:

```
(if (not (member item month-list))
  (push item month-list)
```

This pushes *item* onto *month-list* only if it isn't already on the list.

The following function takes a list that may contain repeated elements and returns that list containing only one instance of each item.

5–1 build–unique–list

```
(defun build–unique–list (original–list)
  (let (item unique–list)
    ; initialize the unique list
    (setq unique–list nil)
    ; loop through list
    ; get an item by popping the original–list
    (while (setq item (pop original–list))
      ; if the item isn't on unique–list, then push it on it
      (if (not (member item unique–list))
        (push item unique–list)))
  ; return the unique list
  unique–list
  ))
  ; try it out
  (setq my–list (build–unique–list (list 1 2 3 2 3 3 4 5 7)))
  ; returns (7 5 4 3 2 1)
```

Function 5–12: assoc

(assoc *item list–of–cons &optional test*)

Returns*: first cons whose *car* matches *item

E.g. *(setq n (list (cons 1 2) (cons 3 4) (cons 5 6)))*
Returns ((1 . 2) (3 .4) (5 . 6))
(assoc 3 n)
Returns (3 . 4)

Remember *cons*? *Assoc* provides a powerful way to retrieve information from a list of *cons*es. It compares an item to the *car* of every set of *cons*es on a list. If it matches, it returns the *cons*.

As with *member*, you can substitute your own test.

Function 5–13: rassoc

(rassoc *item list–of–cons &optional test)*

Returns*: first cons whose *cdr* matches *item

E.g. *(setq n (list (cons 1 2) (cons 3 4) (cons 5 6)))*
Returns ((1 . 2) (3 .4) (5 . 6))
(rassoc 4 n)
Returns (3 . 4)

Rassoc is exactly the same as *assoc* except it looks at the *cdr* of each *cons*, rather than the *car.*

Function 5–14: rplaca

(rplaca *list new–cons)*

***Returns*: the modified *list* with *new–cons* replacing the original car**

E.g. *(setq n (list (cons 1 2) (cons 3 4) (cons 5 6)))*
Returns ((1 . 2) (3 .4) (5 . 6))
(rplaca n (cons 7 8))
Returns ((7 . 8) (3 . 4) (5 . 6))

Rplaca alters a list by replacing the *car* with a new item.

Rplaca works on any list, not just lists of *cons*es, because I-Lisp in a sense considers all lists to be lists of *cons*es — the *car* is the first half of the *cons*, and everything else (the *cdr*) is the second half. For example:

(setq n (list 1 2 3 4))
Returns (1 2 3 4)
(rplaca n 6)
Returns (6 2 3 4)

Function 5–15: rplacd

(rplacd *list new–cons)*

Returns*: the modified *list

E.g. *(setq n (cons 1 2))*
Returns (1 . 2)
(rplacd n 3)
Returns (1 . 3)

Rplacd is just like *rplaca* except that it replaces the *cdr*, not the *cons.*

By the way, the *d* in *rplacd* is a reminder that it deals with the *cdr*, since *d* is the only letter that differentiates *cdr* from *car.*

Function 5–16: selt

(selt *item list number*)

Replaces item at position *number* in *list* with *item*

E.g. *(setq n (list "a" "b" "c" "d"))*
Returns ("a" "b" "c" "d")
(selt "f" n 2)
Returns "f"; n is now ("a" "b" "f" "d")

Selt is zero-based; it thinks the first element of the list is number 0.

Function 5–17: delete

(delete *item list &optional test*

***Returns*: *list* with *item* deleted**

E.g. *(setq n (list 1 2 3 4 1))*
Returns (1 2 3 4 1)
(delete 1 n)
Returns (2 3 4)

Delete looks at a list and deletes from it everything that matches the specified item.

Function 5–18: list–length

(list–length *list*)

Returns*: number of items on *list

E.g. *(setq n (list 1 2 3))*
Returns (1 2 3)
(list–length n)
Returns 3

This gives you the number of top-level items in the list. (Notice that it is *not* zero-based.) For example:

(setq n (list 1 2 3 (list 4 5)))
Returns (1 2 3 (4 5))
(list–length n)
Returns 4

If you want to count all the items in a list, including the items in lists contained within the list, you could do the following:

5–2 total–list–length

```
(defun total–list–length (lst)
; Count all items in a list, even if the list has lists as items in it
  (let (item)
    (while (setq item (pop lst))          ; get an item from the list
      (if (typep item 'cons)              ; is the item itself a list?
        (total–list–length item)          ; if yes, then invoke this
                                          ; function again
```

```
    (inc *ctr*)))                    ; else, increase the counter
  *ctr*                              ; when done, return the counter
))
; example of a use of this function:
(setq m (list 1 2 3 (list 4 5) 6 7 (list 8 9 (list 10 11 (list 12 13))) 14 (list 15
16)))
; Returns (1 2 3 (4 5) 6 7 (8 9 (10 11 (12 13))) 14 (15 16))
(setq *ctr* 0)                       ; use a global counter ... not pretty!
(setq length (total-list-length m))
Returns 14
```

This is actually a tricky function. You can either use it blindly, or read the next paragraph and find yourselves plunged into the depths of recursion. It's optional.

The *total–list–length* function takes a list as an argument. It *pop*s the first item from the list. Then it asks if the item is itself a list, by using the *typep* function we have not yet explained; *typep* lets you find out if something is of a particular type. If the item is a list, then it invokes *total–list–length* again. It keeps invoking the function again until it has gone all the way to the bottom of nested lists. (The invoking of a function from within that very function is what's known as *recursion*. I-Lisp lets you use recursion effectively.) If the item is not itself a list, then it increases the counter (**ctr**) by one. The function returns **ctr**.

Unfortunately, **ctr** has to be defined outside of the function itself; otherwise it would be reset to 0 each time *total–list–length* is invoked. This means **ctr** is a *global variable;* by convention, global variables are given names that begin and end with asterisks. Global variables are to be avoided whenever possible.

An example — Cards

As an example of ways of processing lists, here is the beginning of a card game. It creates a deck of cards, shuffles it, and then deals it out. (By the way, when you go to deal the cards out to two players, remember you don't have to deal the cards alternately to the two players because the computer's *random* function has already ensured randomness). The following uses several functions we have not yet covered.

5–3 shuffle

```
(defun shuffle (d)
; Shufffle the cards
(let (card1 card2 card1-number card2-number)
  ; get a random card
  (repeat 100        ; swap cards 100 times
    (setq card1-number (random 52))    ; get a random card number
    (setq card2-number (random 52))    ; get another to swap with
    (setq card1 (elt d card1-number))  ; get the card at that number
```

```
    (setq card2 (elt d card2-number))    ; get the other card
    (selt card1 d card2-number)          ; put card1 at card2's position
    (selt card2 d card1-number))         ; and card2 at card1's position
))
```

5–4 create-shuffle-deck

```
(defun create-shuffle-deck()
; create deck and shuffle it
  (let (deck suits suit card)
    (setq deck nil)
    ; create list of suits
    (setq suits (list "Club" "Spade" "Heart" "Diamond"))
    ; create the deck
    (while (setq suit (pop suits))
      (setq card 0)
      (while (<= (inc card) 13)
        (push (cons card suit) deck)
    ))
  ; Take the deck and shuffle it
  (shuffle deck)
deck
))

; try it out
(setq d (create-shuffle-deck))
```

Chapter 6
Control Structures

Interleaf Lisp provides a variety of ways of looking at conditions and then doing one thing or another.

Function 6–1: if

(if *statement consequence* &optional *else*)

***Returns*: If *statement* is *t*, then returns value of *consequence*; else returns value of *else* (if any)**

E.g. *(setq x 2)*
(setq y 3)
(if (= x y)
(stk–open "They're equal!")
(stk–open "They're not equal!"))
Creates stickup and returns value of else (nil)

An *if* statement performs a test, and if the test turns out to be non-nil, then it does what follows. If the test is nil, then it looks for an *else* to do; the *else* is optional.

With *if*, it is crucial to pay attention to the parentheses, for these are what tell you which groups of statements are the test and which are the actions.

The test statement can be complex, involving *and*s, *or*s and *not*s, multiple conditions, and the like. The stuff that is to be done if the test statement is true can be as long and complex as necessary. This is likewise true for what gets done if the test statement is false.

Let's look at some examples:

```
; have user enter name
(setq name (stk–open "Enter your name" :input 40))
; check if name is "Smith"
(if (string= name "Smith")
   ; create stickup and end the if
   (stk–open "Hello, Smith"))
```

Here's another example.

```
; have user enter name
(setq name (stk-open "Enter your name" :input 40))
(if name
  (do-something name))
```

In this case, you prompt the user to enter a name. If the user does, then *name* becomes non-nil and so the function *do-something* (which we haven't bothered defining in this example) gets executed. If, however, the user chooses *Cancel* on the stickup, rather than entering a name, then *name* becomes nil, and *do–something* never gets executed.

Another example:

```
(if (setq name (stk-open "Enter your name" :input 40))
  (do-something name))
```

This does exactly the same thing as the previous example, except it compresses it by remembering that *setq* returns the value of the variable that is being assigned a value. So, in this example, first a stickup is created. Then *name* is set to its value. Then the value is checked to see if it is nil. If it's non–nil, then the test is passed.

Let's expand this example to get an *else*:

```
(if (setq name (stk-open "Enter your name" :input 40))
  (do-something name)
  ; else
  (stk-open "So you prefer to be anonymous"))
```

Notice that in this example, the second line ends with a single right parenthesis, rather than double parentheses. Matching the initial parenthesis says that the *if* statement is done and there is no *else* to be executed. In the most recent example, the parenthesis that matches the initial one is on the last line, closing the *if* statement after the *else* has been noticed.

The general form of an *if* statement is:

```
(if (test)          ; if test is non-nil
  (action1)         ; then do action1
  (action2)         ; else, do action2
)
```

But suppose you do more than just a single line function as the result of an *if* test. You use *progn*:

Function 6–2: progn

(progn *statement1 ...*)

***Returns*: Return of last statement**

E.g. *(progn*
(setq x 6)
(setq y x))
Returns 6

Progn says that there are a series of statements to be executed. For example:

```
(if (setq name (stk-open "Enter your name" :input 40))
  (progn
    (do-something name)
    (do-something-more name)
    (stk-open "Thanks for entering your name.")))
```

This will execute three lines (*do-something*, *do-something-more*, and *stk-open*) if the user enters a name.

You can also use *progn* to wrap a set of statements to execute as your *else* statement. For example:

```
(if (setq name (stk-open "Enter your name" :input 40))
  (progn
    (do-something name)
    (do-something-more name)
    (stk-open "Thanks for entering your name."))
  ; else
  (progn
    (do-anonymous-stuff)
    (setq anonymous t)
    (do-more-anonymous-stuff)))
```

Once again, pay attention to where the parentheses are.

Here is another way *if* could be used.

```
(setq age 14)
(stk-open
    (if (> age 13)
        (setq text "Please continue")
        ; else
        (setq text "Please get your parents before continuing")))
```

Remember that *stk-open* expects some text. In this example, the *if* determines what to *setq* *text* with; the *setq* returns the value *text* is set to; the *if* returns the value of the *setq* statement. And that value is used as the text of *stk-open*.

Function 6-3: unless

(unless *statement then*)

Returns*: If *statement* is t, then returns nil; else returns value of *then

E.g. *(setq x 3)*
(unless (= x 3)
(stk–open "X isn't equal to 3"))
Returns nil; does not create the stickup

Unless is the opposite of *if*. While *if* only executes its *then* statements if the test statement is non-nil, *unless* executes the *then* statements if the test statement is nil.

You could simulate *unless* by using *not* with *if*. For example, the following two expressions do exactly the same thing:

```
(if (not (string= name "smith"))
  (stk–open "Your name isn't Smith")

(unless (string= name "smith")
  (stk–open "Your name isn't Smith"))
```

Function 6–4: cond

(cond *(test consequence)(test consequence) ...)*

***Returns*: Value of last *then* evaluated or nil if none is evaluated**

E.g. *(setq x 5)*
(cond
((= x 4) (setq y 0))
((= x 5) (setq y 1)))
Returns 1

With *cond*, you can test multiple conditions. When one condition is met (the statement is non-nil), then the following *then* is executed, the rest of the *cond* is skipped, and the value of the *then* is returned. The form of a *cond* is:

```
(cond
  ((test 1) (then 1))
  ((test 2) (then 2))
)
```

Here's an example:

```
; let user enter skill level
(setq degree (stk–open "Enter highest degree earned:" :input 5))
; assign skill level based on degree
(cond
  ; if degree is BA or BS, set skill to 1
  ((or (string= degree "BA") (string= degree "BS"))      (setq skill 1))
  ; if degree is MA, set skill to 2
  ((string= degree "MA")                                 (setq skill 2))
  ; if degree is Ph.D., set skill to 3
  ((string= degree "Ph.D.")                              (setq skill 3))
  ; in all other instances, set skill to 0
  (t                                                     (setq skill 0)))
```

There are a couple of points worth noting in this example. First, the first condition in this example is an *or* statement. You can make the conditions as complex as you want, so long as they are all contained within a single set of parentheses. Second, the last condition is *t*. This is to handle the case in which none of the prior tests are matched. The *t* is always going to be evaluated as non-nil, so the *then* that follows it is going to be executed *if* the *cond* gets that far ... which it will so long as none of the previous conditions are met. So, the last line only gets executed if none of the previous ones do. If you didn't use this "trick," and none of the conditions were non-nil, then the *cond* would return nil. (Remember, 0 and nil are not the same things.)

Function 6–5: while

(while *statement then*)

***Returns*: value of last statement evaluated**

E.g. *(setq x 0)*
(while (< x 3)
(inc x))
Returns 0

While lets you do something repeatedly until some test comes up nil. For example:

```
(setq parts-list (list 12 45 23))          ; make list with 3 numbers in it
(while (setq item (pop parts-list))        ; get next on list
  (enter-into-order-form item)             ; do something with the part number
  (send-update-to-inventory item))         ; do something else with it
```

Remember that *pop* returns the first item on a list, and removes that item from the list. So, by *setq*ing *item*, we put the first element of *parts–list* (12) into *item*, and shorten *parts–list* by one. And, since *setq* returns the value of what has been *setq*ed, the *while* gets a non–nil value ... until the last value has been *pop*ped off of *parts–list*. So, when *parts–list* no longer has any members, *pop*ping it results in a nil, *item* is *setq*ed to nil, the entire test returns nil, and the *while* loop grinds to an end.

You can see that *while* can be very useful when, for example, looking at every component in a document. Perhaps you want to find all the components named "para" and turn them into "bullet," or check to see if there are any components with a particular attribute value. With *while*, you can go through the entire document, component by component, until there are no more components. The following example shows this, although parts must remain mysterious until later chapters:

6–1 add–footnote

```
(defun add-footnote ()
; Creates footnote frame at end of any component named quote.
; Expects a frame master named footnote to already exist in the
; document.
(let (cmpn)
  ; get first component
```

```
(setq cmpn (tell *document* mid:get-child))
; while not out of cmpns
(while cmpn
  ; is cmpn named "quote"?
  (if (string= "quote" (tell cmpn mid:get-name))
    (progn
      ; get marker at end of cmpn
      (setq marker (tell cmpn mid:get-marker t))
      ; go to marker
      (doc-goto-marker marker)
      ; create frame named "footnote"
      (tell *text-editor* mid:create :frame "footnote")))
  ; get next cmpn unless we're done
  (setq cmpn (tell cmpn mid:get-next)))
))

; try it out
(add-footnote)
```

This looks at every component in a document, and adds a footnote frame at the end of any component named "quote." (It expects you to have already created a master for a frame called "footnote.")

Function 6–6: repeat

(repeat *number statement*)

***Returns*: value of last statement executed**

E.g.
```
(setq x 2)
(repeat 3
        (setq x (* x x)))
Returns 256
```

Repeat does exactly the same thing as using a counter to count the number of iterations of a *while* loop, but does so more economically. The following two approaches are equivalent:

```
(setq ctr 5)
(setq test 0)
(while (not (zerop ctr))
  (dec ctr)  ; decrements ctr by one
  (inc test))

(setq test 0)
  (repeat 5
  (inc test))
```

(In the above examples, *(inc test)* is just a sample of something you might want to do.)

Function 6–7: catch and throw

Catch and *throw* provide a way to exit from a loop before the test condition has become nil. For example:

6–2 find–first–cmpn–of–name

```
(defun find-first-cmpn-of-name (cmpn-name)
; Finds first component named cmpn-name
(let (cmpn (found-cmpn nil))
  ; get first component
  (setq cmpn (tell *document* mid:get-child))
  ; set place to jump to with throw
  (catch 'done
  ; while not out of cmpns
  (while cmpn
    ; matches requested name?
    (if (string= cmpn-name (tell cmpn mid:get-name))
      ; if so, save cmpn ...
      (progn
        (setq found-cmpn cmpn)
        ; ... and exit loop
        (throw 'done))
      ; else get next cmpn unless we're done
      (setq cmpn (tell cmpn mid:get-next)))))
    ; Return the cmpn
  cmpn
  ))
```

In the previous example, the function takes the name of a component as its argument. It looks through all the components for one with the same name. If it finds one, it saves the component in the variable *found–cmpn* and *throw*s itself out of the loop entirely. Otherwise, it loops through all of the components and returns nil.

If the system comes across a *throw*, even if it is in the middle of a loop that by all rights isn't yet done, it will throw control back to the *catch*. Notice that both *catch* and *throw* take a symbol as their argument — you should make up a name that is appropriate and put a quote in front of it.

Function 6–8: toplevel

If the system comes across a *throw*, it *throw*s control back to the *catch*; if it comes across a *toplevel*, it exits the I-Lisp program entirely.

Function 6–9: quit

Quit is like *toplevel*. It exits the I-Lisp program immediately.

There are two sets of functions — advanced topics — that allow you to iterate through a list without having to use any control structure such as *while* or *repeat.*

Function 6–10: apply

(apply *function args*)

Applies *function* to the *args*, the last one of which must be a *list*

E.g. *(apply '+ (list 2 3 4))*
Returns 9

Apply takes a function and applies it to each member of a list (or lists). In the above example, the plus function is repeatedly applied to a list, with the answer accumulating. (Notice that the function has to be prefaced by a single quote to indicate that it is not to be evaluated immediately.)

The following example prints at the text caret a list of every component in a document, followed by the page it begins on. It does this by building a list of the component names and page numbers, and then using *apply* to insert the names and numbers repeatedly into the document.

6–3 show–cmpn–names

```
(defun show-cmpn-names ()
; creates list in a doc of cmpns and page numbers
  (let (c name pagenumber (cmpn-name-list nil))
  ; notice that this let statement sets cmpn-name-list to nil
    ; loop through the document, looking at each cmpn
    ; get the first cmpn (the child of the document object))
    (setq c (tell *document* mid:get-child))
    (while c
      ; get the name
      (setq name (tell c mid:get-name))
      ; get the page the cmpn is on, get the page's number, and
      ; convert that number to an ascii string
      (setq pagenumber
        (itoa (doc-page-number-of (doc-page-of c))))
      ; push the info onto list in form "name:pagenumber"
      ; followed by a return represented by "\n"
      (push (concat name ":" pagenumber "\n") cmpn-name-list)
      ; get the next cmpn and continue the loop
      (setq c (tell c mid:get-next))
    )
    ; we've pushed the list into reverse order, so unreverse it now
    (setq cmpn-name-list (reverse cmpn-name-list))
    ; repeatedly tell the current spot in doc to insert contents the list
    (apply #'tell (doc-point-marker) mid:insert
```

cmpn–name–list)

))

Function 6–11: mapcar

(mapcar *#function list* &optional *more–lists)*

Performs *function* on successive cars of *list* and *more–lists*; returns a list of the results

E.g. *(mapcar #'+ (list 1 2 3) (list 4 5 6))*
Returns (5 7 9)

If the lists are of different lengths, *mapcar* quits after completing the shortest list. (The # is Lisp magic. It needs to be there. Just use it blindly.)

Here are some more examples:

```
(mapcar #'> (list 4 6 2 7) (list 9 7 1))
Returns (nil nil t)
; make a function to check if half of one is greater than twice the other
(defun half–greater–than–twice (a b)
  (> (/ a 2) (* b 2)))
(mapcar #'half–greater–than–twice (list 20 30 40) (list 4 8 25))
Returns (t nil nil) because half of twice 20 is greater than twice 4, etc.
```

These capabilities allow you to put together complex and powerful applications with Interleaf Lisp.

Chapter 7
Numbers and Characters

Interleaf Lisp provides powerful ways of manipulating numbers and characters. In this chapter you'll learn some of those techniques.

Types of numbers

I-Lisp recognizes two types of numbers: integers and floats. Integers are numbers with no decimal places. Floats (or floating point numbers) are numbers with decimal places. Some functions only work with one or the other type of number, so keeping them straight can be important.

Integers have to be between –2,147,483,648 and 2,147,483,647. Floats don't suffer from this rather severe restriction.

Fortunately, you can convert from one type to the other.

Function 7–1: float

(float *integer*)

Returns float equivalent of *integer*

E.g. *(float 34)*
Returns 34.0

Function 7–2: ceiling

(ceiling *number*)

Returns smallest integer equal to or greater than *number*

E.g. *(ceiling 3.01)*
Returns 4

Function 7–3: floor

(floor *number*)

Returns smallest integer equal to or less than *number*

E.g. *(floor 3.99)*
Returns 3

Function 7–4: truncate

(truncate *number*)

Returns integer of *number* with nothing after its decimal point

E.g. *(truncate 3.99)*
Returns 3

Function 7–5: round

(round *number*)

Returns closest integer (rounds up or down); if .5, rounds up to nearest integer

E.g. *(round 2.51)*
Returns 3

The differences among these are more obvious if you look at some examples:

(ceiling 34.6)
Returns 35
(floor 34.6)
Returns 34
(truncate 34.6)
Returns 34
(round x)
Returns 35

The difference between *floor* and *truncate* becomes apparent when you have negative numbers. For example:

(floor -43.6)
Returns −44
(truncate)
Returns −43

Testing numbers

You may want to find out what type of number you're dealing with, or, more frequently, test the relationship of two numbers. Interleaf Lisp provides a useful set of functions for doing so — many of which may not look like functions because they are denoted by a single character, e.g., "+" or ">."

Function 7–6: floatp

(floatp *number*)

Returns t if *number* is a float; else, nil

E.g. *(setq x 3.0)*
(floatp x)
Returns t

Function 7–7: integerp

(integer *number*)

Returns t if *number* is an integer; else, nil

E.g. *(setq x 3.0)*
(integerp x)
Returns nil

Function 7–8: >

(> *number1 number2*)

Returns t if *number1* is greater than *number2*

E.g. *(> 3 2.1)*
Returns t

Function 7–9: <

(< *number1 number2*)

Returns t if *number1* is less than *number2*

E.g. *(< 3 2.1)*
Returns nil

Function 7–10: >=

(>= *number1 number2*)

Returns t if *number1* is greater than or equal to *number2*

E.g. *(>= 3 2.1)*
Returns t

Function 7–11: <=

(<= *number1 number2*

Returns t if *number1* is less than or equal to *number2*

E.g. *(<= 3 2.1)*
Returns nil

Function 7–12: =

(= *number1 number2*

Returns t if *number1* is equal to *number2*

E.g. *(= 3 2.1)*
Returns nil

All of these tests are straightforward.

Hint: To translate these expressions into our normal format, imagine the operator moving into the middle position. So, for example, (> x y) is the same as (x > y).

Function 7–13: zerop

(zerop *number*)

Returns t if *number* is zero; else, nil

E.g. *(setq x 0.1)*
(zerop x)
Returns nil

This is a briefer way of saying (= 0 *number*).

Function 7–14: evenp

(evenp *number*)

Returns t if *number* is even; nil if odd

E.g. *(setq x 3)*
(evenp x)
Returns nil

Function 7–15: oddp

(oddp *number*)

Returns t if *number* is odd; else, nil

E.g. *(setq x 3)*
(oddp x)
Returns t

Altering numbers

In addition to the arithmetical operators we'll discuss in the next section, I-Lisp lets you alter numbers in some especially useful ways.

Function 7–16: dec

(dec *integer*)

Reduces (decrements) *integer* by one and returns the decreased number

E.g. *(dec 15)*
Returns 14

Function 7–17: inc

(inc *integer*)

Increases *integer* by one and returns the increased number

E.g. *(inc 15)*
Returns 16

Inc and *dec* are both one-step ways of adding or subtracting one from a number. This is especially useful when you are using a variable as a counter. For example:

7–1 count–open–docs

```
(defun count-open-docs (container)
(let (kids kid count)
```

```
    ; set counter to 0
    (setq count 0)
    ; get all children of a container
    (setq kids (dt-children container))
    ; count them
    (while (setq kid (pop kids))
      ; only counts if it's a document and it's open
      (if (and (is-of-class kid dt-document-class)
        (tell kid mid:get-props :opened))
          (inc count)))
  ; return the counter
  count
  ))
```

The *inc* and *dec* functions could be easily replaced by slightly more cumbersome code. For example:

```
(setq x (- x 1))
```

and

```
(setq x (1- x))
```

are both the same as

```
(dec x)
```

Arithmetical operations

Interleaf Lisp uses *prefix notation* for arithmetical operations — the arithmetical operators come first. So, for example, in I-Lisp, you add 3 and 4 with the expression:

```
(+ 3 4)
```

And you subtract 3 from 4 with:

```
(- 4 3)
```

With this in mind, the following arithmetical operators are available to you in I-Lisp:

Function 7–18: +, –, /, *

(+ *number number*)

Returns result of operations

E.g. *(+ 3.12 5.6 7.8)*
Returns 16.52
(– 65 7.3)
Returns 57.7
(3 4.2)*
Returns 12.6
(/ 6 3)
Returns 2

You can, of course, nest operations within parentheses. For example:

```
(+ 1 (– 5 2.2))
Returns 3.8
(+ 1 (– 2 (* 3 (/ 4.65 5))))
Returns 0.21
```

Notice that +, –, and * can take more than one argument. The + and * do the obvious thing: they keep adding or multiplying the arguments. On the other hand, – does something a little less predictable; it in effect subtracts from the first argument the sum of all the remaining arguments. For example:

```
(– 10 1 2 3)
Returns 4
```

Note that / returns a float so long as either of its arguments is a float. If, however, they are both integers, / returns only the integer portion of the result. For example:

```
(/ 12.0 5)
Returns 2.4
(/ 12 5)
Returns 2
```

Failure to keep this in mind is an excellent way to create a recalcitrant bug.

Function 7–19: mod

(mod *number1 number2*)

Returns the remainder of dividing *number1* by *number2*

E.g. *(mod 12 5)*
Returns 2

Function 7–20: abs

(abs *number*)

Returns the absolute value of *number*

E.g. *(abs –3.12)*
Returns 3.12

In addition, there are some mathematical functions available to you. Their operation and results should need no elucidation at this point:

sqrt	*square root*
sin	*sine*
cos	*cosine*
tan	*tangent*
asin	*arc sine*

acos *arc cosine*

atan *arc tangent*

This functionality can be built on to provide powerful mathematical processing capabilities.

Characters

A single character for I-Lisp is not the same thing as a string with one character in it. A single character is expressed by prefacing it with a pound sign and a backslash. For example:

#\A	uppercase A
#\Z	uppercase Z
#\;	semicolon

There are some characters with special meaning:

#\\n	newline, or linefeed
#\\t	tab

Characters are represented internally by a code number, which usually maps to the ASCII code numbers, although Interleaf provides an extended set of characters above 127 on the ASCII chart.

Function 7–21: character

(character *argument*)

Returns*: character equivalent to *argument

E.g. *(character 65)*
#\A

You can use *character* to convert a code number or a single-element string into a character. For example:

(character "A")
Returns #\A

Function 7–22: char–int

(char–int *char*)

Returns*: code number of *char

E.g. *(char–int #\A)*
Returns 65

Function 7–23: int–char

(int–char *number*)

***Returns*: character represented by *number* in Interleaf character code**

E.g. *(int–char 65)*
Returns #\A

These two functions allow you to go back and forth between characters and integers.

Here is a function that will print a list of selected character codes; it prints it into whatever component your insertion caret is in when you invoke the function. It gives you the character code in base 10 and in hexadecimal, as well as the character itself. (This function uses some terms we have not yet introduced, including *doc–point–cmpn*, *get–marker*, *concat*, and *itoa*. The *format* function will be explained later in this chapter.)

7–2 print–char–codes

```
(defun print–char–codes (min max)
; prints char codes number followed by char for numbers MIN through
; MAX.
; E.g., (print–char–codes 65 91)
  (let (marker)
    (while (<= min max)
      ; get marker at end of cmpn
      (setq marker (tell (doc–point–cmpn) mid:get–marker t))
      ; build string with concat
      (tell marker mid:insert (format nil "~D\t~X\t~A\n" min min (int–char
min)))
      ; increment character
      (inc min))
))
```

For example, *(print–char–codes 32 127)* would put the following into your document:*

32	20	
33	21	!
34	22	"
35	23	#
36	24	$
37	25	%
38	26	&
39	27	'
40	28	(
41	29	)
42	2A	*
43	2B	+

*When you run this function you may notice some upside-down question marks where you expected a character to be. The question marks are Interleaf's way of saying that there is no printable character for that character code. You may also notice that what follows character 10 is a blank line. That isn't a mistake. Character 10 is the line feed character, i.e., the character that causes a new line to be inserted into a document.

44	2C	,
45	2D	–
46	2E	.
47	2F	/
48	30	0
49	31	1
50	32	2
51	33	3
52	34	4
53	35	5
54	36	6
55	37	7
56	38	8
57	39	9
58	3A	:
59	3B	;
60	3C	<
61	3D	=
62	3E	>
63	3F	?
64	40	@
65	41	A
66	42	B
67	43	C
68	44	D
69	45	E
70	46	F
71	47	G
72	48	H
73	49	I
74	4A	J
75	4B	K
76	4C	L
77	4D	M
78	4E	N
79	4F	O
80	50	P
81	51	Q
82	52	R
83	53	S
84	54	T

85	55	U
86	56	V
87	57	W
88	58	X
89	59	Y
90	5A	Z
91	5B	[
92	5C	\
93	5D	]
94	5E	^
95	5F	_
96	60	`
97	61	a
98	62	b
99	63	c
100	64	d
101	65	e
102	66	f
103	67	g
104	68	h
105	69	i
106	6A	j
107	6B	k
108	6C	l
109	6D	m
110	6E	n
111	6F	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7A	z
123	7B	{
124	7C	|
125	7D	}

126	7E	~
127	7F	"

Notice that although character 32 looks blank, it's actually the space character. Unrecognized characters are printed as ¿. And if you were to print *(print–char–codes 262 266)* you'd get the special hyphen and dash characters as well.

262	106	-
263	107	–
264	108	¿
265	109	¿
266	10A	—

If you are in a document and need to know a character's code, simply select it and execute *props* from the text-selected menu. That will put the character code into the message line of the document window. (It will show it as a hexadecimal number.)

Function 7–24: char–downcase

(char–downcase *character*)

***Returns*: lowercase equivalent of *character*, or *character* if no lower case**

E.g. *(char–downcase #\A)*
Returns #\a

Function 7–25: char–upcase

(char–upcase *character*)

***Returns*: uppercase equivalent of *character*, or *character* if no upper case**

E.g. *(char–upcase #\a)*
Returns #\A

These allow you to switch from upper to lower case and vice versa. Where there is no upper (or lower) case, the character itself is returned. For example:

(char–downcase #\a)
Returns #\a
(char–upcase #\3)
Returns #\3

Function 7–26: plain–char–p

(plain–char–p *argument*)

***Returns*: t if *argument* is a character; else, nil**

E.g. *(plain–char–p "This is a string")*
Returns an error because it is a string, not a character

You can test whether what you have is a character by using *plain–char–p*. If it has a character code between 0 and 255, it will return t.

(plain–charp (int–char 65))
Returns t

(plain−charp (int−char 265))
Returns nil
(plain−charp "A")
Returns error message because "A" is a string, not a character

Strings

I-Lisp comes with powerful functions for handling strings, as you might expect given that Interleaf 5 itself is often used as a sort of super word processor.

Function 7–27: string

(string *argument*)

Returns*: string made from *argument

E.g. *(string 'Alabama)*
Returns "Alabama"

String turns a single character or a symbol into a string.

String will turn a character into a string. For example:

(string #\A)
Returns "A"

Function 7–28: concat

(concat *string1* string2. . .)

***Returns*: string consisting of *string1* plus rest of string arguments**

E.g. *(concat "Welcome to" " Hawaii")*
Returns "Welcome to Hawaii"

This is a very useful function because it allows you to create strings by using variables. For example:

7–3 name–into–stickup

```
(defun name−into−stickup ()
  (let (name msg)
    ; get name
    (setq name (stk−open "Enter your name" :input 40))
    ; concat it into message
    (setq msg (concat "You said your name is " name "!"))
    (stk−open msg)))
```

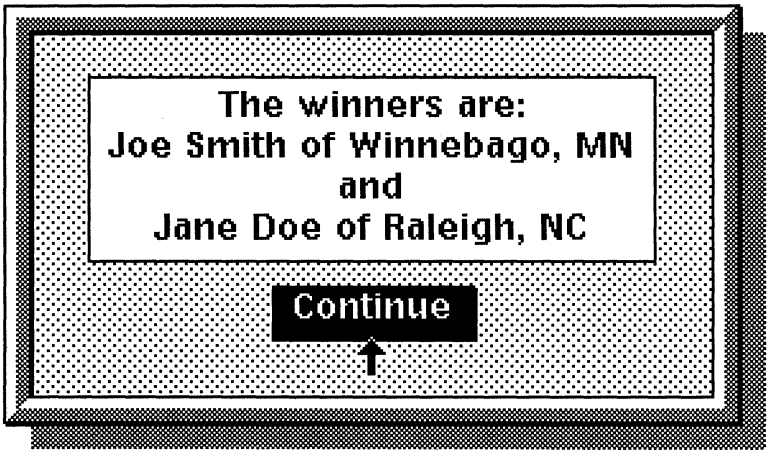
(You could have skipped the second-to-last line and replaced the last line with *(stk−open (concat "You said your name is " name "!"))*.)

Concat is actually quite powerful. For example:

```
; create a list for this example
(setq winners
```

```
    (list
      (list "Joe Smith" "Winnebago, MN")
      (list "Jane Doe" "Raleigh, NC")))
  ; concat the winners and make stick-up
  (stk-open
  (concat "The winners are:\n"
    (car (setq winner (pop winners)))
    " of " (car (cdr winner)) "\nand\n"
    (car (setq winner (pop winners)))
    " of " (car (cdr winner))))
```

Creates stickup that says:



(The *\n* indicates a new line. You could also just put in the number 10, which is the number of the new-line "character.")

The function *format* gives you much more control over the printed format of strings, as we will discuss at the end of this chapter.

*Function 7–29: *case–sensitive**

The system comes with a global variable, **case–sensitive**, which initially is set to nil. If you set it to t, then some of the string comparison functions will report two strings as not equal if they differ only in terms of upper or lower case letters. So, "polish" and "Polish" would be reported as not equivalent.

You can alter this by the line *(setq *case-sensitive* t)* or *(setq *case-sensitive* nil)*

Function 7–30: string=

(string= *string1 string2*)

Returns*: t if *string1* has the same content as *string2

E.g. *(string="Cary Grant" "Archibald Leach")*
Returns nil

We discussed *string=* in Chapter 5. The global variable, **case–sensitive** does not affect this comparison; *string=* is always case-sensitive.

Function 7–31: string–contained
(string–contained *string1 string2*)
***Returns*: position where *string1* is contained within *string2;* else, nil**
E.g. *(string–contained "leaf" "Interleaf")*
Returns 5

String–contained tells you where one string begins in another. (The first letter is counted as zero.) This function is controlled by the variable **case–sensitive**, although it can be overridden.

You can have *string–contained* begin looking at a particular spot, count backwards or forwards, and be case sensitive or not, all by using some optional arguments. For example:

```
(string–contained"A" "012a456")
Returns 3
; begin at 4th position
(string–contained"A" "012a456" 4)
Returns nil
; begin at 3rd position
(string–contained"A" "012a456" 3)
Returns 3
; make it case sensitive
(string–contained "A" "012a3456" 0 :forward t)
Returns nil
; make it not case sensitive
(string–contained "A" "012a3456" 0 :forward nil)
Returns 3
```

If you do not specify whether the search should be case sensitive or not, the case sensitivity is whatever **case–sensitive** has been set to.

You can use *string–contained* to check whether a string (or character) is part of a known set. For example:

7–4 is–digit

```
(defun is–digit (c)
; returns the position of the char in the string of numbers if char is a digit,
; or else a nil
  (string–contained (string c) "0123456789"))
```

This takes a character, converts to a string, and looks for it in the string "0123456789." If it's there, then, then the character is a digit. We could the same thing for a grade:

7–5 is–grade

```
(defun is–grade (c)
```

```
; returns t if char is an academic grade A–F
  (string–contained (string (char–upcase c)) "ABCDF"))
```

Of course, we are not limited to checking characters; we can also check strings:

```
7–6 is–grade–plus–or–minus
(defun is–grade–plus–or–minus (s)
; returns number if s is an academic grade A–F with plus or minus
  (string–contained s "A+ A– B+ B– C+ C– D+ D– F"))
```

Notice, by the way, that Interleaf distinguishes among several types of dashes. Not only are there "-", "–" and "—", but Interleaf's own Lisp font (used in its documentation to show Lisp code) has its own dash. So, be careful of how you're expressing your minus sign when using this function.

Function 7–32: string–length

(string–length *string*)

Returns*: the number of characters in *string

E.g. *(string–length "12345")*
Returns 5

Notice that *string–length* is *not* zero-based.

Function 7–33: string–uppercase

(string–uppercase *string*)

Returns*: uppercased version of *string

E.g. *(string–uppercase "aBc1 DeF")*
Returns "ABC1 DEF"

Function 7–34: string–lowercase

(string–lowercase *string*)

Returns*: lowercased version of *string

E.g. *(string–lowercase "aBc1 DeF")*
Returns "abc1 def"

Function 7–35: substring

(substring *string number1 number2*)

***Returns*: subset of *string* starting at position *number1* and continuing for *number2* characters**

E.g. *(substring "abcdefg" 2 3)*
Returns "cde"

Notice that the *position number* is zero-based.

Function 7–36: string–left–trim

(string–left–trim *string1 string2*)

***Returns*: *string2* with any characters in *string1* removed from its left**

E.g. *(string–left–trim "–$+" "+$100")*
Returns "100"

Function 7–37: string–right–trim

(string–right–trim *string1 string2*)

***Returns*: *string2* with any characters in *string1* removed from its right**

E.g. *(string–right–trim "–+" "$100+")*
Returns "$100"

These functions allow you to snip off the beginning or ending of a string. For example, you might want to remove leading or trailing spaces; these functions make it easy.

If you want to trim characters from both the beginning and ending simultaneously, use the following function:

Function 7–38: string–trim

(string–trim *string1 string2*)

***Returns*: *string2* with any characters in *string1* removed from its left and right**

E.g. *(string–trim "<>!" ">>!!100 !!<<")*
Returns "100"

The following functions will help you convert numbers to strings.

Function 7–39: atoi

(atoi *string*)

Returns*: integer of string version of number expressed by *string

E.g. *(atoi "62")*
Returns 62

Function 7–40: itoa

(itoa *integer*)

Returns*: string version of *integer

E.g. *(itoa 62)*
Returns "62"

Function 7–41: atof

(atof *string*)

Returns*: float version of *string

E.g. *(atof "62.1")*
Returns 62.10

Function 7–42: ftoa

(ftoa *float*)

Returns*: string version of *float

E.g. *(ftoa 62.1)*
Returns "62.10"

These *"ascii–to–number"* and *"number–to–ascii"* functions convert a string to a number, or a number to a string.

These are very useful functions since you will frequently be encountering numbers written as characters in documents. And most stickups return strings, so if you want to have a user type a number into a stickup, you will probably be using *atoi* to convert it to a number, and *itoa* to convert it back to something printable. For example:

7–7 string–or–number–stkup

```
(defun string–or–number–stkup ()
; stickup handles either text or numbers
(let (text number)
  (setq text (stk–open "Enter product name or number" :input 50))
  (if (setq number (atof text))
    ; it's a number, so
    ; do something ... for demo, just give a confirming stickup
    (stk–open "It's a number!")
    ; else it's a string, not a number
    (stk–open "Text, text, text."))
))
```

Sample program: Leaf of Fortune

Here is a sample program that uses many of the string functions we've looked at so far. It's a bit like the traditional *hangman* game, although it doesn't draw the graphic of the person about to be hanged; consider adding the graphic to be one of your first challenges after you've looked at the chapter on Interleaf graphics.

```
; create a list of phrases, as a global variable
(setq *leaf–words* (list "TURN OVER A NEW LEAF" "LEIF ERIKSON"))
```

7–8 get–a–saying

```
(defun get–a–saying ()
; choose from list of sayings, or quit if there aren't any left
(let (phrase which)
  ; if there are any words on the list ...
  (if *leaf–words*
    (progn
      ; get a random number
      (setq which (random (list–length *leaf–words*)))
      ; get phrase from the list
      (setq phrase (nth which *leaf–words*))
      ; remove the selection from the list
      (setq *leaf–words* (delete phrase *leaf–words* :test 'string=))
```

```
      )
  ; else
    (progn
      (stk-open "Out of phrases. Replenish and try again!")
      (quit)))
; return the phrase
phrase
  ))
```

7-9 take-a-guess

```
(defun take-a-guess ()
 (let (ch)
  (setq ch (stk-open "Enter a character to guess" :input 1))
 ; if no response, then quit
 (if (not ch)
   (quit))
(string-uppercase ch) ; uppercase it
))
```

7-10 get-char-s

```
(defun get-char-s (s n)
; returns char in string s at position n
(substring s n 1))
```

7-11 build-guess

```
(defun build-guess (s)
 ; make a copy, with asterisks, of the saying to be guessed
  (let ( (blank-guess "")
    (x 0)
      (s-len (string-length s)))
    ; look at each char until we finish the string
    (while (< x s-len)
      ; get next char in saying
      (setq c (get-char-s s x))
       ; increment counter
      (inc x)
      ; if char's not a blank, then put in an asterisk
      (if (not (string= c " "))
        (setq blank-guess (concat blank-guess "*"))
      ; else
      (setq blank-guess (concat blank-guess " "))))
  blank-guess
))
```

7–12 is–guess–in–saying

```
(defun is–guess–in–saying (c s)
; loops through s building list of all places where there's a c
  (let ( (spot–list nil)
         (spot 0))
  ; loop until no more guessed–chars in the string
  (while (setq spot (string–contained c s spot))
    ; make list of all places where guessed–char is in string
    (push spot spot–list)
    ; increment spot counter so we keep looking for next
    ; guessed–char
    (inc spot))
spot–list
))
```

7–13 put–char–s

```
(defun put–char–s (s c n)
; Put char into string at position n
(let (s1 s2)
 (setq s1 (substring s 0 n))                    ; get first half
 (setq s2 (substring s (+ 1 n) (string–length s))) ; get 2nd half
  (concat s1 c s2)                ; join 2 halves w/new char 'tween
))
```

```
;–––––––––––––––––––––––––––– Run this to play the game
```

7–14 leaf–of–fortune–game

```
(defun play–leaf–of–fortune ()
  (let (saying
        new–cmpn
        guess
        letter–guessed
        spot location–list
        marker
        turns
        max–turns)
  ; set maximum number of turns
  (setq max–turns 20)
  (setq turns 0) ; set counter

  ; get a saying
  (setq saying (get–a–saying))

  ; build a blank guess with asterisks and underscores
```

```
  (setq guess (build-guess saying))

  ; put the display of letters to turn
  ; needs cmpn master named "response" to already be defined
  (setq new-cmpn (tell *cmpn-editor* mid:create "response"))
  ; create cmpn
  (tell (tell new-cmpn mid:get-marker t) mid:insert guess)
  ; update the display
  (doc-flush-queue)              ; force a redisplay of the doc

  ; loop to play the game until use up maximum turns or get successful
guess
  (while (and (<= turns max-turns)
    (not (string= guess saying)))
      (setq letter-guessed (take-a-guess))    ; get a guessed letter
      (inc turns)                             ; increase turn counter
        (if letter-guessed                    ; if not cancel out of stickup
          (progn
          ; create new cmpn
      (setq new-cmpn (tell *cmpn-editor* mid:create "response"))
      ; is guessed letter in saying?
      (if (setq location-list
        (is-guess-in-saying letter-guessed saying))
            (progn
            ; take list of positions, substitute letter guessed for *
            (while (setq spot (pop location-list))
              (setq guess
                (put-char-s guess letter-guessed spot)))))
        ; insert new display
        (tell (tell new-cmpn mid:get-marker t) mid:insert guess)
        (doc-flush-queue) ; force a redisplay of the doc
    )))

  ; check to see if we have a winner
  (if (<= turns max-turns)
    (progn
      (stk-open "You won!"))
    ; else
    (stk-open "Better luck next time"))
))
; try it
(play-leaf-of-fortune)
```

Format

Function 7–43: format

A special function, called *format*, has enough options and parameters to deserve its own small section. *Format* will output your strings in just about any way you can imagine. All you have to do is learn the magic incantations.

The basic format of *format* is:

(format *destination control–string string*)

The *destination* is usually going to be nil, which will send the string to the standard output. You could, however, specify a specific stream (see *streams* in Chapter 17) for the output.

The *control–string* is a special set of characters used by *format* to figure out how to format the *string* you want formatted. The control–string begins with a tilde ("~") and ends with a character. Between the two can be more special codes to further specify how you want the string formatted. For example:

```
(format nil "~D" 12345)
Returns "12345"
(format nil "~:D" 12345)
Returns "12,345"
(format nil "~@:D" 12345)
Returns "+12,345"
(format nil "~R" 12345)
Returns "twelve thousand three hundred forty five"
```

These codes can themselves be part of a string to be printed. For example:

```
(format nil "The answer is ~:D" 12345)
Returns "The answer is 12,345"
(format nil "The characters ~R are the same as ~:D" 1023 1023)
Returns "The characters one thousand twenty–three are the same as
1,023"
```

Notice that in that last example, we had to write "1023" twice; each of the codes looks for its own argument at the end of the statement.

Format allows you to force a string to fill a certain size field by *padding* any leftover spaces with your choice of characters. You do this by specifying the size of the field (default 0), the minimum number of characters added when padding (default 0), the number of times a character is repeated (default 1), and the character you want to use for padding (default space). For example:

```
(format nil "~10,1,1A" "one")                ; default pad char
Returns "one       "
(format nil "~10,1,1,'#A" "one")             ; # as pad char
Returns "one#######"
```

```
(format nil "~10,1,1,'#@A" "one")          ; pad on left, not right
Returns "#######one"
```

Notice that you have to put a single quotation mark before the character you want to use as the pad character.

Here are the codes:

A

Prints strings

@ (right justifies)

```
(format nil "Welcome to ~A" "NY")
Returns "Welcome to NY"
(format nil "~15,1,1,'.@A" "Smith")     ; right-justified with dots as pad char
Returns ".....Smith"
```

B

Prints in base 2 (binary) format

: add commas

@ print number's sign even if positive

```
(format nil "~B" 34)
Returns "10010"
(format nil "~:B" 34)                   ; add comma
Returns "10,010"
```

D

Prints integers in base 10

: add commas

@ print number's sign even if positive

```
(format nil "~10,'*@:D" 12345 )         ; right-justified with * pad chars
Returns "***+12,345"
(format nil "~10,'*D" 12345 )           ; right-justified with * pad chars
Returns "12345*****"
```

E

Prints a floating point number in exponential form

@ print number's sign even if positive

```
(format nil "~E" 123456.78)
Returns "1.2345678e5"
```

F

Prints a floating point number.

@ print number's sign even if positive

```
(format nil "~F" 12)
Returns " 12.0"
(format nil "~6F" 12)                   ; force width of 6, including decimal point
Returns " 12.0"
```

(format nil "~6,2F" 12.3456) ; width of 6, 2 on right of decimal point
Returns " 12.35"
(format nil "~6,4@F" 1200.345678)
Returns "+1200.3457"

G
Prints number in floating point notation if it isn't too large, and exponential format if it is very large. See *E* and *F* for formatting rules
(format nil "~G" 10000000000000.01)
Returns "10000000000000.01"
(format nil "~G" 100000000000000.01) ; add one more zero ...
Returns "1.e+14"

O
Prints integer in base 8 (octal)
: prints comma
@ print number's sign even if positive
(format nil "~O" 44)
Returns "54"

P
Lets you stay grammatical by printing "s" if its argument isn't the number one
: reuses previous argument
@ prints "y" if argument is 1, "ies" otherwise
(format nil "Knocked on ~D door~P" 1 1)
Returns "Knocked on 1 door"
(format nil "Knocked on ~D door~P" 2 2)
Returns "Knocked on 2 doors"
(format nil "Knocked on ~D door~:P" 3) ; only one argument, used twice
Returns "Knocked on 3 doors"
(format nil "Knocked on ~R door~:P" 300) ; Use ~R, not ~D
Returns "Knocked on three hundred doors"
(format nil "Ate ~R french fr~@P" 1 1)
Returns "Ate one french fry"
(format nil "Ate ~R french fr~@P" 12 12)
Returns "Ate twelve french fries"

R (no parameters)
By itself, R prints an English equivalent of a number
: prints name as ordinal number
@ prints number as Roman numeral
(format nil "~R" 23)
Returns "twenty–three"
(format nil "~:R" 23)

Returns "twenty–third"
(format nil "~@R" 23)
Returns "XXIII"

R (parameters)
If you give R parameters, it prints a number in the base (radix) of your choice
: prints commas
@ prints number's sign even if positive
(format nil "~6R" 1000) ; base 6
Returns "4344"
(format nil "~7:R" 1000) ; base 7, with commas
Returns "2,626"
(format nil "~17R" 1000) ; base 17
Returns error: radix out of range

T
Prints a tab
(format nil "NY~8TAlbany\n~8TNYC") ; tab of 8 spaces
Returns "NY Albany
NYC"

X
Prints in base 16 (hexadecimal)
; prints commas
@ prints number's sign even if positive
(format nil "~X" 100)
Returns "64"
(format nil "~:@X" 10000)
Returns "+2,710"

$
Print in dollars and cents format
: number's sign before padding
@ number's sign even if positive
(format nil "~$" 12.3456)
Returns "12.35"
(format nil "~$" 12.1)
Returns "12.10"
(format nil "~@$" 123.3456) ; add plus sign
Returns "+123.35"
(format nil "$~$" 123.3456) ; add $ sign
Returns "$123.35"

%
Prints newline characters

```
(format nil "Skip two lines~2%and start again")
```
Returns *"Skip two lines*

and start again"

Take *format* slowly and you'll learn to love it.

Chapter 8
The Desktop

Traditional Interleaf 5 provides its own graphical user interface which uses the desktop metaphor. Some versions, however, now use another windowing system's desktop — for example, the Motif version of Interleaf 6 has shipped and versions for Microsoft Windows and NT are under development as this book is being written. In this chapter, we'll examine ways in which the traditional Interleaf 5 desktop can be manipulated by Interleaf Lisp. This includes operating on desktop objects — opening them, closing them, etc.

Desktop objects

The Interleaf 5 desktop is hierarchical. At the highest level — the parent of all parents — is the desktop itself. It can contain containers of various sorts, such as folders, cabinets and books. These containers have their own methods. For example, books know how to index their contents. Then there are a variety of non-container objects, such as documents, image icons, lisp scripts, and palette icons. Of course containers may be nested any way the user wants. In addition, there are special objects such as the dictionary, clipboard, profile drawer, system cabinet, and custom cabinet. Finally, there may be new classes of objects created by other developers; for example, someone might make a "card file" document with special functionality and make it into a new class of object, or a "vault" container that cannot be opened without first entering the proper password.

Getting Desktop Objects

As with other objects, a desktop object's name is a property of the object and must not be confused with the object itself. So, to address a desktop object, you have to first get ahold of it somehow. Interleaf Lisp provides a variety of techniques for doing so. Some of the important objects that come with the system have their own variables:

dt-desktop — current desktop
dt-clipboard — the clipboard

In other cases, there are functions which easily provide access to particular objects.

Function 8–1: dt–find–profile
(dt–find–profile)
Returns: the profile script container
Function 8–2: dt–find–system
(dt–find–system)
Returns: the System5 cabinet
Function 8–3: dt–find–custom
(dt–find–custom &optional argument)
Returns: Custom cabinet, Selection cabinet or No Selection cabinet
E.g. *(dt–find–custom nil)*
Returns the No Selection Cabinet

If you supply no argument, *dt–find–custom* returns the Custom cabinet. An argument of nil returns the No Selection cabinet. An argument of non-nil returns the Selection cabinet.

When you use one of these functions and look at what gets returned, it will look something like *#<dt-obj 0x71f000>*. The code number is used for internal purposes. (It's actually the memory address where the object begins.) About the only good it does you is that if, while debugging, you notice two objects have the same code, you can be pretty sure they are the same object.

Of course you need to be able to get at any object, not just the ones that come with the system:

Function 8–4: dt–object
(dt–object name)
Returns: desktop object with that name
E.g. *(dt–object "myfile.doc")*
Returns the object named "myfile.doc"

You can also specify a pathname. That pathname can be relative to the desktop or not. For example, if you have a document whose full pathname is "/u/serviette/chris/desktop/bigbook.boo/5CardDraw.doc," you could use either of the following two expressions:

```
(dt–object "/u/serviette/chris/desktop/bigbook.boo/5CardDraw.doc")
(dt–object "bigbook.boo/5CardDraw.doc")
```

Notice that the desktop-relative pathname does not begin with a slash. Also be aware that pathnames may look different in your operating system. Finally, notice that we use the file name as it appears to the operating system, which may be different from the way it appears on the desktop; most operating systems, for example, don't like spaces to be part of file names, and some don't like filenames to be longer than 8+3 characters; Interleaf 5 lets you get around this limitations, but *dt–object* wants the operating system name. If you have a desktop object and want to get its pathname, you can use:

Function 8–5: dt–path

(dt–path object &key)

Returns: pathname to object

E.g. *(dt–path mydoc :part :backup)*
Returns "bigbook.boo/mydoc.doc,1"

If you provide the *:root* keyword with an argument of nil, you'll get just the filename as it appears to the operating system. If you give it the key *:part*, it will give you the pathname to the object's "part" files. These are files that contain information related to the main document. The Interleaf software automatically maintains these files invisibly to the end user. For example, if a user cuts, copies, moves or renames a file on the desktop, all the associated parts files will be handled appropriately. There are several different sorts of part file:

Keyword	Description	UNIX
:main	main document	–
:backup	backup file	,1
:checkpoint	created when a checkpoint save is done	,2
:crash	created when user chooses "file" from crash menu	,3
:work–in–progress	work–in–progress file	,4
:methods	Interleaf Lisp applied to the main document	,5
:saved–data	data placed on document through Lisp	,6
:image–summary	linked image file information	,7
:autonumber–summary	information about autonumbers	,8
:index–summary	information about the main document's index	,9
:attribute	attribute information	.@ (prefix)
:parts	Lisp scripts	,21-9

Matters are different with Interleaf 5 for DOS. Because of the eight-character (and three character extension) limitation imposed by DOS, Interleaf 5 uses a different scheme for denoting the different types of files. (Of course on the Interleaf 5 desktop, the user can use the usual 31-character file names.) It "hashes" the names so that the operating system name may look quite different from what's on the desktop.

Containers	Folders, cabinets, drawers and books end with a distinctive 3-character code: Folders: fdr Cabinets: cab Drawers:drw Books: boo E.g., "System 5" cabinet is "systecab" when looked at in DOS
Documents	For the main document (the one that shows up as an icon), I5 provides a .doc extension. Catalogs have a .sty extension The parts files that have ",number" extension in Unix instead have a .d#number extension. So, a Unix file called "Mydoc.doc,5" would show up in DOS as "mydoc.d#5." When the part number is double digit, I5 switches to letters. So the Unix "Mydoc.doc,21" becomes the DOS "mydoc.d#l" and "Mydoc.doc,22" becomes "mydoc.d#m"
Lisp files	.lsp extension

The following function will put into a stickup the file system name of any selected icon. (It uses a function we have not yet explained.)

8–1 show–dos–name–stkup

```
(defun show-dos-name-stkup ()
  ; make sure we're in the right container
  (dt-set-container (dt-pointer-container))
  (stk-open (dt-path (dt-child-selected))))
```

The following does the same but puts the name into a stayup (using a function we haven't yet explained either), so you can refer to it as you do other things.

8–2 show–dos–name–stayup

```
(defun show-dos-name-stayup ()
; make sure we're in the right container
  (dt-set-container (dt-pointer-container))
  (stayup (dt-path (dt-child-selected))))
```

Put this into your Selection cabinet if you plan on using it regularly.

The following function returns the name as it appears on the desktop.

Function 8–6: get–name

(tell object mid:get–name)

Returns: object's desktop name

E.g. *(tell object mid:get–name)*
Returns "MyWork"

Notice that this does not return the extension, e.g., it returns "MyWork," not "MyWork.doc."

You might want to get at the selected icon. But, with Interleaf 5, you might have two icons selected, each in its own container (a folder, cabinet, etc.). You can specify the working directory.

Function 8–7: dt–get–container
(dt–get–container)
Returns: current container
Function 8–8: dt–set–container
(dt–set–container container)
Sets container
E.g. *(dt–set–container *dt–desktop*)*
Returns the desktop

Suppose you have several directory windows open at once on the desktop. The current container, unlike the current document, is not necessarily the one the mouse pointer was in most recently (and whose header changed color to indicate this). It is, roughly, the container you most recently worked in. But the notion of "working in" can be complex. So, to work in a container programmatically — for example, to open a document within it — you will usually want to force the container to be the current container. Here is one technique.

8–3 make–current–container

```
(defun make–current–container (dtobj)
; sets parent of dtobj to be the current container
  (let (container)
    ; get container of dtobj
    (setq container (tell dtob mid:get–parent))
    ; set container
    (dt–set–container container) ))
```

If you want the container the mouse pointer is currently in, the following will get it for you:

Function 8–9: dt–pointer–container
(dt–pointer–container)
***Returns*: container mouse is currently in, or nil if mouse isn't in a container**
Function 8–10: dt–child–match
(dt–child–match container &optional *name class*)
Returns: desktop object of *name* in *container* of type *class*
E.g. *(dt–child–match mybook "mydoc" dt–document–class)*

This will look in the container *mybook* (which is a desktop object, not a name) for a document named "mydoc." If you do not specify a name, *dt–child–match* will return the first

object in the container. If you do not specify class, it will return all objects of every type in the container.

The notion of the "first" or "last" objects in a container is problematic. Usually, the system understands the order of objects in a container the way human beings do: "First" means at the top and to the left. Sometimes, however, the order is different. So don't count on being able to predict the order.

You can look at every object in a container:

Function 8–11: get–child

(tell *container* mid:get–child)

Returns: first object in *container*, or nil if no objects in it

E.g. *(tell mybook mid:get–child)*
Returns first document in the mybook container object

Once you have one child, you can get all the others.

Function 8–12: get–next

(tell *object* mid:get–next)

Returns: next child, or nil if no other objects

E.g. *(tell mydoc mid:get–next)*
Returns next object in mydoc's container

Interleaf 5 includes functions for getting only the selected children or non-selected children of a container:

Function 8–13: dt–child–not–selected

(dt–child–not–selected &optional *container*)

Returns: first unselected child in *container*; if all selected, nil

Function 8–14: dt–child–selected

(dt–child–selected &optional *container*)

Returns: first selected child in *container*; if none selected, nil

Since you may have multiple files selected, all in different containers, this function gives you the first one selected in whatever is the current container (or in the container object specified as an argument). Since it often isn't obvious what the current container is, you are well advised to specify the container whenever possible.

Hint: You may well find yourself using *dt–child–selected* (and its variants) quite frequently. As a shortcut, you could load the following function:

8–4 get–sel

```
(defun get–sel ()
; returns the selected object and posts name in a stickup to confirm
  (if (not (dt–child–selected))
```

```
    (stk-open (concat "No selected object in "
      (tell (dt-get-container) mid:get-name)))
  ; else, if something is selected
    (progn
      ; optionally, post name in stickup
      (stk-open (tell (dt-child-selected) mid:get-name)) ;
      (dt-child-selected))) ; return selected object
)
; try it
(setq doc (get-sel))
```

You can get all the contents in one fell swoop:

Function 8–15: dt–children

(dt–children &optional container argument)

Returns: all contents of a container

E.g. *(dt–children mybook :selected)*
Returns list of all selected contents

The arguments are *:selected*, *:not–selected,* or *:all*. If you don't specify any arguments, it will default to giving you all.

Here is a function that checks a selected document to see if it has a backup file.

8–5 check–for–backup

```
(defun check-for-backup ()
; checks selected doc to see if there is a backup file
(let (doc (backup-doc nil))
  (setq doc (dt-child-selected))
  ; is there a selection?
  (if (not doc)
      (stk-open "Nothing selected.")
    ; else
    (progn
      ; get path to backup
      (setq backup-doc (dt-path doc :part :backup))
      ; is there a file with that name?
      (if (probe-file backup-doc)
        (stk-open "There is a backup.")
        ; else
        (stk-open "No backup."))))
  backup-doc
))
```

This function uses *probe-file*, which we have not discussed. *Probe-file* returns t if there's a file with that pathname, and nil if there isn't.

The following checks *all* the children of a container and builds a list of those that have no backup file.

8–6 check–for–backups

```
(defun check–for–backups (container)
; checks all of container's children for backup files
(let (doc–list doc (no–backup–list nil) (counter 0))
  ; get all the children
    (setq doc–list (dt–children container))
    ; check each for backup
    (while (setq doc (pop doc–list))
      ; is it a doc? (uses a function not yet discussed)
      (if (is–of–class doc dt–document–class)
        (progn
          ; increase counter
          (inc counter)
          ; is there backup?
          (if (not (probe–file (dt–path doc :part :backup)))
              ; put it on list
              (push doc no–backup–list)))))
    ; report the number of no backups
    (stk–open (format nil "Of ~D documents, ~D had no backup."
      counter (list–length no–backup–list)))
  ; return the list
    no–backup–list ))
```

Here are two brief functions that will look through a container and find every child, even containers nested within the original container. It goes to the current open document and creates a separate line for each document name, with a tab for each level of hierarchy. In essence, the following functions will build an outline view of the contents of a container. (They use other functions not yet discussed.)

8–7 is–container

```
(defun is–container (obj)
; returns t if obj is a container
  (is–of–class obj dt–container–class)
)
```

8–8 got–a–child

```
(defun got–a–child (obj lev)
; creates line and inserts name of obj
  (let ( m (text ""))
    ;get a marker
    (setq m (doc–point–marker))
```

```
    ; create string, one tab for each successive level
    (repeat lev
      (setq text (concat text "\t")))
    (setq text (concat text (tell obj mid:get-name) "\n"))
    ; move marker down to next line
    ; first get a marker at the end of the line
    ; and then move it one token ahead
    (setq m (tell (doc-point-line) mid:get-marker t) )
    (tell m mid:move-by 1)
    ; go to marker
    (doc-goto-marker m)
    ; insert the text
    (tell m mid:insert text)
  ))
```

8–9 get–children

```
(defun get-children (container level)
; recurses through container getting all children,
; and sending each to got-a-child
  (let (children child)
    (setq children (dt-children container))
    (while (setq child (pop children))
      ; is it a container?
      (if (is-container child)
        (progn
          (inc level)
          (got-a-child child level)
          (get-children child level))) ; recurse
      ; else not a container
        (progn
          (got-a-child child level)))
    (dec level)
))

  ; give it a try, with the Create cabinet as our sample
  (get-children (dt-find-create) 0)
```

The function *get–children* gets all the children in a container, looks at each, and sends it to *got–a–child,* which constructs a new line. If *get–children* comes across a child which is itself a container, it recurses, i.e., it calls itself from within itself and passes itself the child that is a container. *Get–children* then looks at all of the children of the new container, and so forth, until it has examined every single child.

Sometimes an end user is trying to access a document buried within several layers of containers. Usually a user gets there by opening each container, each of which makes its own

window on screen. The following function opens an object and simultaneously closes its parent, keeping the user's screen cleaner. (This uses the *mid:open* method, which we haven't yet discussed.)

8–10 open–and–close

```
(defun open-and-close ()
;Opens selected item and closes its parent, unless the parent is the
desktop.
(let (obj parent)
    ; set container to where the mouse now is
    (dt-set-container (dt-pointer-container))
    ; anything selected?
    (if (not (setq obj (dt-child-selected)))
      (stk-open "Nothing selected.")
      ; else - something is selected
      (progn
        ; make it unselected
        (tell obj mid:set-props :selected nil)
        ; open it
        (tell obj mid:open)
        ; close the parent, unless it's the desktop
        (if (not (equal *dt-desktop* (setq parent (tell obj mid:get-parent))))
          (tell parent mid:close))))
))

; try it out, binding desktop ^O
(kbd-bind kbd-dt-map "\^o" 'open-and-close)
```

If you use this method of opening a container, you may want to open all the parents until you reach the desktop.

8–11 open–parents

```
(defun open-parents ()
; opens parent of selected item
(let (container parent)
    ; set container to where the mouse now is
    (dt-set-container (dt-pointer-container))
    (setq container (dt-get-container))
    ; get parent
    (setq parent (tell container mid:get-parent))
    ; open it
    (tell parent mid:open)
))
```

```
; try it out, binding desktop ^C
(kbd-bind kbd-dt-map "\^c" 'open-parents)
```

The following will close the current container and open its parent.

8–12 open-parent-and-close-current

```
(defun open-parent-and-close-current ()
; opens parent of selected item
(let (container parent)
    ; set container to where the mouse is now
    (dt-set-container (dt-pointer-container))
    (setq container (dt-get-container))
    ; get parent
    (setq parent (tell container mid:get-parent))
    ; if not desktop
    (if (not (equal *dt-desktop* container))
      (progn
        ; close current
        (tell container mid:close)
        ; open parent
        (tell parent mid:open)))
))

; try it out, binding desktop ^A
(kbd-bind kbd-dt-map "\^a" 'open-parent-and-close-current)
```

Properties

Now that we can get desktop objects, we can get and set their properties, which we do, of course, by *tell*ing them to set their properties.

Function 8–16: get-props

(tell *dtobj* mid:get-props &optional &keyword)

Returns specified properties of *dtobj*

E.g. *(tell (dt-child-selected) mid:get-props :opened)*
Returns t if the selected obj is open

Function 8–17: set-props

(tell *dtobj* mid:set-props &keyword *value*)

Sets specified property of dtobj to value

E.g. *(tell (dt-child-selected) mid:set-props :save-default :ascii)*
Sets the default save method of the selected object to ascii

The properties you can get or set are:

Argument	Return
:window–position	Cons of x and y coordinates of upper left edge of the window position, in screen pixels
:window–size	Cons of width and height of window in screen pixels
:icon–position	Cons of x and y coordinates of upper left edge of icon
:icon–size	Cons of width and height of icon
:opened	t if icon is open, nil if not
:selected	t if icon is selected, nil if not
:autopositioned	t if icon is autopositioned, nil if not
:save–default	Format to save document in (:fast, :ascii or nil for inherit)
:attributes–control	Lists document attributes
:catalog–exports	List of what a catalog exports (:components, :auto–numbers, :frames, :diagramming–objects, :tables, :page–properties, :headers–footers)

Let's look at one example. If you use your operating system to copy a document into a book, the operating system doesn't do the "bookkeeping" Interleaf requires to know where that document is supposed to be in the set of chapters composing the book. So, the next time you open the book icon, the system will report that there is an autopositioned document in it. Unfortunately, it doesn't tell you which is the autopositioned icon. Here is a script that reports this to you.

8–13 autoposition–report

```
(defun autoposition-report ()
; reports which selected docs were auto-positioned
(let (book (doc-list nil) doc (auto-list nil) (names ""))
   ; get the book
   (setq book (dt-child-selected))
   ; make sure it's a book
   (if (not (is-of-class book dt-book-class))
     (progn
       (stk-open "Not a book.")
       (quit)))
   ; get list of docs
   (setq doc-list (dt-children book))
   ; check each one
   (while (setq doc (pop doc-list))
     (if (tell doc mid:get-props :autopositioned)
       (progn
         ; build list
         (push doc auto-list)
         ; build string of names of autopositioned docs
         (setq names
```

```
            (concat names (tell doc mid:get-name) 10))))
    )
    ; report
    (if auto-list
      (progn
        (stk-open (concat "Autopositioned icons:" 10 names)))
      ; else
      (stk-open "No autopositioned icons"))
  ; return list of autopositioned icons
  auto-list
))
```

One reason we had this function return the list of autopositioned icons is so we can automate the next step:

8–14 de–autoposition

```
(defun de-autoposition ()
; makes any selected docs that were auto-positioned not auto-positioned
  (let (doc (a-list nil))
    ; get the list of autopositioned icons
    ; if any, then do something
    (if a-list
      (progn
        (setq a-list (autoposition-report))
        ; select them
        (while (setq doc (pop a-list))
          (tell doc mid:select))
        ; open the container - lisp seems to like this
        (tell (dt-child-selected) mid:open)
        ; do a book sync of all selected documents
        (dt-sync-selected (dt-child-selected))))
))
```

The following script is a little unusual in that it has to be run by using the Load function on your Custom menu.

8–15 jump–script

```
(defun jump-script (doc)
; makes this script's icon jump up and back
(let (pos new-pos )
    ; get the current icon position
    (setq pos (tell doc mid:get-props :icon-position))
    ; set a new icon position 15 pixels higher
    (setq new-pos
```

```
      (cons
        (car pos)
        (- (cdr pos) 15)))
  ; move icon 15 pixels up
  (tell doc mid:set-props :icon-position new-pos)
  ; restore it to its original position
  (tell doc mid:set-props :icon-position pos)
))

; now try it out, using this very icon
(jump-script (dt-script))
```

When you *load* this script, it will jump up 15 pixels and then settle down again. It gets the current icon position, which is returned as a *cons* of the left/right axis and the top/bottom axis. It then subtracts 15 from the *cdr* of the *cons*, sets the icon position to the new height, and then returns it to the original position.

When this *defun* gets called by this script, the object it is passed is the actual script itself. That's the purpose of *dt-script*:

Function 8-18: dt-script

(dt-script)

***Returns*: the desktop object of the script currently being interpreted**

The following script uses *jump-script* to make all the objects in a container jump up and down, in sequence.

8-16 wave

```
(defun wave (container)
; makes all docs in current window jump and down once in wave-like
fashion
; uses jump-script
(let (child)
  ; get first child
  (setq child (tell container mid:get-child))
  ; go through the children
  (while child
    ; make the child jump
    (jump-script child)
    ; get next child
  (setq child (tell child mid:get-next)))
))
```

You will notice that there is a very short delay after each jump and possibly the sound of disk access. That's because Interleaf 5 has to save the icon position information for each icon in a

separate file. The ability to adjust the position of the icons is actually quite important in a system such as Interleaf 5 where the visual arrangement of documents is considered a crucial part of the information about those documents. For example, the order of documents in a book window determines the order of the chapters in the book.

Function 8–19: dt–create

(dt–create *name* &optional :type *icon–type*)

Returns*: new desktop object of type *icon–type* and name *name

E.g. *(dt–create "newfile" :type 'book)*
Returns new book object

Dt–create allows you to create instances of a class. Assuming that the class has no contents as part of its definition, the objects you create also will have no contents. But they will have methods: if you create a book, for example, the popup menu when a document is selected will show all the usual book methods (indexing, TOC, etc.). (If you leave off the word *:type*, the system will create a plain host file, i.e., an empty file icon that shows up on the desktop as computer paper.)

Function 8–20: dt–search–position

(dt–search–position *old–position container)*

Returns*: a cons for a new position for an icon in *container* near *old–position

E.g. *(setq old–pos (tell dtobj mid:get–props :icon–position))*
Returns (638 . 224)
(dt–search–position old–pos (dt–get–container))
Returns (654 . 240)

If you tell it that the old position is *(cons 0 0)*, it will give you the first available position. The following creates a new icon in the first open spot in a container:

8–17 dt–create–and–position

```
(defun dt–create–and–position (icon–name icon–type container)
; creates an icon in the named container
  (let (obj new–pos)
    ;set container
    (dt–set–container container)
    ; create object
    (setq obj (dt–create icon–name :type icon–type))
    ; get new position
    (setq new–pos (dt–search–position (cons 0 0) container))
    ; move icon to new position
    (tell obj mid:set–props :icon–position new–pos)
    ))
```

You may well want to position an icon as the last in a book, however. The following will do that:

8–18 position–icon–as–last

```
(defun position–icon–as–last (obj container)
; positions obj as last icon in container
  (let (children last–child old–pos new–pos)
    ; set the container
    (dt–set–container container)
    ; get list of all children
    (setq children (dt–children container))
    ; get last child in container
    (setq last–child (car (last children)))
    ; get position of last–child
    (setq old–pos (tell last–child mid:get–props :icon–position))
    ; gets new position
    (setq new–pos (dt–search–position old–pos container))
    ; move icon to new position
    (tell obj mid:set–props :icon–position new–pos)
))
```

Function 8–21: dt–checkpoint

(dt–checkpoint)

Checkpoints all open documents and returns t

This forces a checkpoint save (i.e., an automatic save of a document).

Function 8–22: dt–refresh

(dt–refresh)

Redraws the screen and all open windows, and returns nil

This does what a user does when she selects *Refresh* from the desktop menu.

Function 8–23: dt–sync–document

(dt–sync–document *object* &optional *:save*)

Does a sync on the object; if *:save*, then resaves it also

E.g. *(dt–sync–document dtobj :save)*

Performs a *sync*, updating all the book information about a file.

Function 8–24: dt–sync–selected

(dt–sync–selected *container* &optional *:save*)

Does a sync on all the objects selected in *container*; if *:save*, then resaves it also

E.g. *(dt–sync–container my–book :save)*
Does a sync on all the selected documents in my–book

This syncs all the selected documents at once.

Function 8–25: dt–line–up

(dt–line–up *container*)

Lines up selected objects in *container*

This does within any container what the *Line up* choice allows an end user to do to the selected icons within a book: it aligns the icons, spaces them evenly, and generally improves the scenery.

Function 8–26: toc–create–selected

(toc–create–selected *container*)

Creates table of contents of selected documents within *container*

To create a table of contents for all the objects within a container, you could use the following:

8–19 create–toc–for–all

```
(defun create-toc-for-all (container)
; creates a TOC for all documents in container
  (let (children child)
    ; get all the unselected children
    (setq children (dt-children container :not-selected))
    ; select them
    (while (setq child (pop children))
      (tell child mid:set-props :selected t))
    ; create table of contents for all children
    (toc-create-selected container)
))
```

Function 8–27: dt–size–container–to–contents

(dt–size–container–to–contents *container*)

Sizes *container* to its contents

E.g. *(dt–size–container–to–contents (dt–get–container))*
Sizes current container to its contents and returns cons of window size in pixels

This command redraws a container window so that it just fits the contents of the container.

The following will resize a window to its contents whenever you press ^S on the desktop:

8–20 size–container–to–contents

```
(kbd-bind kbd-dt-map "\^s"
  '(dt-size-container-to-contents (dt-pointer-container)))
```

Function 8–28: print

(tell *dtobject* mid:print *printer-name*)

Prints *dtobject* to printer *printer-name*

The following will give you a list of printers so you can find the printer name.

Function 8–29: *printers*

printers

***Returns*: list of installed printers**

The following builds a stickup of all printers and lets the user choose one. (This function may not work in your operating system environment.)

8–21 choose–printer

```
(defun choose–printer ()
(let (item name–list printers choice names p–list net–name
  (name–string "")
  (done nil)
    (ctr 0))

  (setq printers *printers*)
  (if (not printers)
  (progn
    (stk–open "No printers found")
      (toplevel)))

  ; build list of names
  (setq p–list (copy–list printers))
  (while (setq item (pop p–list)) ; item is one printer's plist
  (inc ctr)
  ; build a name with a number in front of it
  (setq name–string
    (concat name–string
      (itoa ctr)
      ". "
      (fourth item)
      10)))
  ; display list and make choice
  (while (not done)
    (setq choice "")
    (setq choice (stk–open (concat "PRINTERS" 10
      "Enter number of printer" 10
      name–string) :input 3 :left–justify))
  (if (not choice)
    (quit))
  (if (<= (atoi choice) ctr)
    (setq done t))
  ; ok choice
```

```
(if (not done)
  (stk-open (concat "Choose number between 1 and " (itoa ctr))))
(if (not choice)
  (toplevel))
; look up number on printer list to get net name
(setq net-name (second (nth (1- (atoi choice)) printers))))

; return netname
net-name
))
```

Function 8–30: copy

(tell *dtobject* mid:copy *parent* &optional *:link*)

Returns a copy of *dtobject* and places it in container *parent*

E.g. *(tell doc mid:copy *dt-desktop*)*
Creates copy on desktop of selected document

Surprisingly, the copy that is returned by this function is not visible until you tell a container to insert the copy, as explained in the next function.

If you specify *:link* followed by *:simple*, *:original*, or *:container*, the copy will be a linked copy to the document, to the original object to which the document is linked, or to the document's container.

Function 8–31: insert

(tell *container* mid:insert *dtobject)*

Inserts *dtobject* into *container* and deletes *dtobject* from its current container

Returns *t* if it succeeds, *nil* if it doesn't

E.g. *(tell myfolderobject mid:insert doc)*
Cuts myfolderobject from current container and inserts it into this-folder

As mentioned immediately above, if you make a copy of a desktop object, *insert* simply inserts it into the container object. If the desktop object is already inserted (i.e., is visible on the desktop), the object is cut from where it is and is copied into the container object.

Here is a general-pupose utility for copying a file, giving it a new name, and placing its icon in a suitable place on a container.

8–22 copy-and-name

```
(defun copy-and-name (doc new-name container)
; copies doc into container and gives it new-name
  (let (doc-copy new-pos)
    ; make the copy
    (setq doc-copy (tell doc mid:copy container))
```

```
    ; give it a new name
    (dt-set-property doc-copy :icon-name new-name)
    ; insert it
    (tell container mid:insert doc-copy)
    ; get next available spot in window
    (setq new-pos (dt-search-position (cons 0 0) container ))
    ; move it to that spot
    (tell doc-copy mid:set-props :icon-position new-pos)
))

; try it ... make a copy of mydoc.doc, rename it, and put it into current
container
(copy-and-name
  (dt-object "mydoc.doc")
    "mydoc-newname" (dt-get-container))
; Returns cons of new document's position
```

Suppose you want to cut a document from a folder and insert it into your Create cabinet where it will serve as a template.

8–23 move–new–template

```
(defun move-new-template (doc)
; cuts and pastes doc into the Create.cab
  (let (create-cab)
    ; find the create cabinet
    (setq create-cab (dt-find-create))
    ; cut the doc
    (tell doc mid:cut)
    ; do the move - cut doc and move it into Create.cab
    (tell create-cab mid:insert doc)
    ))
```

You might want to add some code (using *dt–search–position*) to put the newly-created icon into the next available spot in the Create cabinet. (See *dt–create–and–position.*)

Function 8–32: delete

(tell *container* mid:delete *dtobject*)

Deletes *dtobject* from *container*

Returns *t* if successful, *nil* otherwise

E.g. *(tell my-book mid:delete (dt-child-selected))*
Deletes selected icon from my-book container

This deletes not only the icon but also any backup, crash and Lisp files. This does *not* put a copy of the file onto the clipboard; there is no recovering the file after you delete it.

Function 8–33: open

(tell *object* mid:open &optional *keywords*)

Opens object according to manner specified by *keywords* and returns *t*

E.g. *(tell (dt–child–selected) mid:open :hide–side–bar t)*
Opens selected document with component bar hidden and returns t

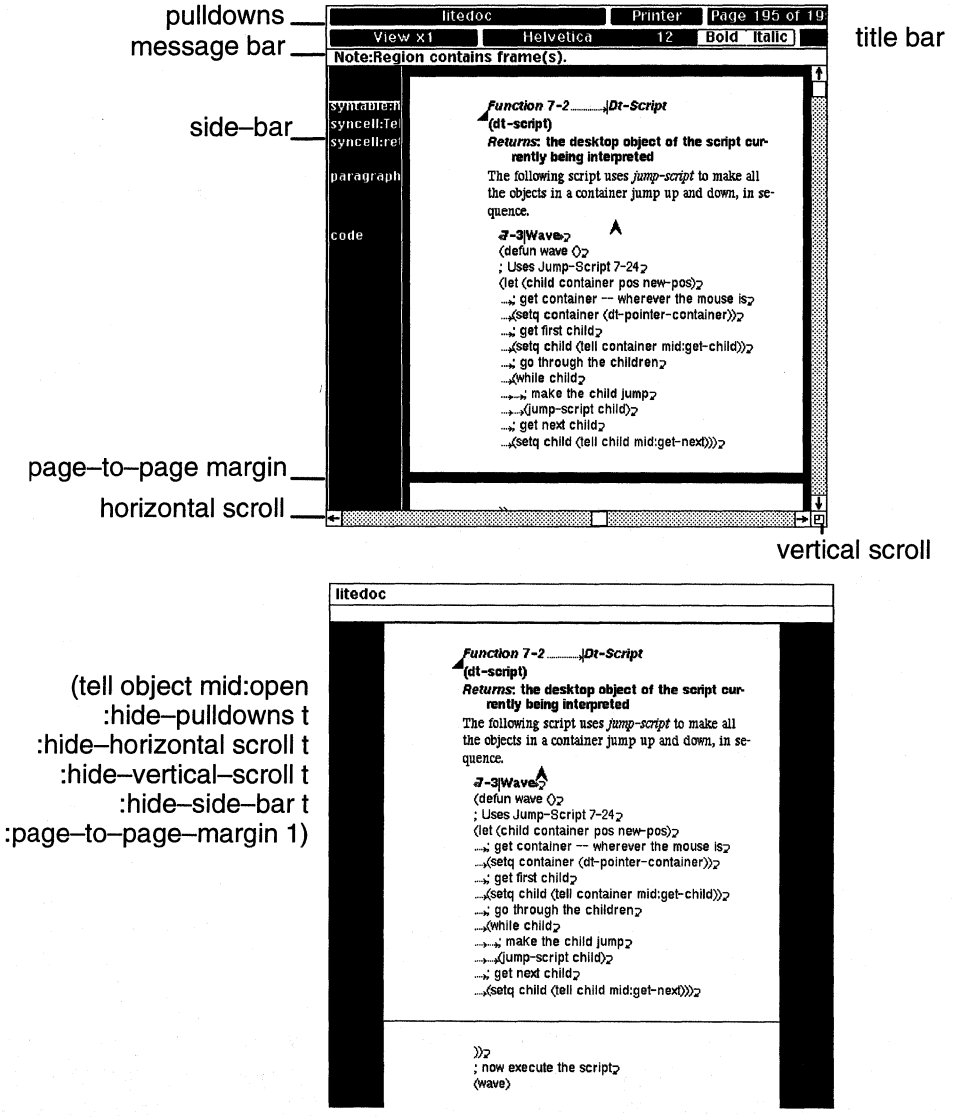
The keywords are:

Keyword	Argument	Comment	Works if already open
:page	#	Open to specified page number	y
:hide–window	t	Opens but is not visible to user	y
:hide–pulldowns	t	Hides header boxes	n
:hide–side–bar	t	Hides component bar	n
:hide–horizontal–scroll	t	Hides righthand scroll bar	n
:hide–vertical–scroll	t	Hides bottom scroll bar	n
:hide–message–bar	t	Hides message bar	n
:hide–title–bar	t	Hides bar with icon name if window opened without header boxes	n
:page–to–page–margin	#	Size of black border between pages in a window	n

In the table's *Arg* column, *t* indicates that it takes a *t* or *nil* as an argument; a # means it takes a number. The following illustrations shows a document opened with the usual parameters and one opened using some of the keywords listed above.

Opening a document that is already open on the desktop will cause it to come in front of any windows that may be obscuring it; this is what happens when the user opens an icon that is already open. As the table shows, some of the keywords will take effect when opening an already-opened document; others will not. You cannot, for example, turn the headers on and then off. For that you would first have to close the document and then open it the way you like.

One of the keywords that does have an effect on an open document is *:page*. This allows you to display various pages very easily. The page number is the page number the user would see. For example, if you were in Chapter 2 and wanted to go to page 22, you would say *(tell doc mid:open :page 22)*, and not have to worry about whether 22 was an absolute page number or an offset.



There are a set of desktop variables you can get and set.

Function 8–34: dt–get–vars

(dt–get–vars &optional *keywords*)

***Returns*: value of all variables, or of *keywords* if specififed**

E.g. *(dt–get–vars :rescan–enabled :rescan–min–idle)*
Return (:rescan–enabled t :rescan–min–idle 3)

Function 8–35: dt–set–vars

(dt–set–vars &optional *keyword value*)

Sets *keyword* variable to *value*, and returns value of last *value* set

E.g. *(dt–set–vars :rescan–enabled nil :rescan–min–idle 4)*
Returns 4

Some of the variables available to you are:

Keyword	Comment	Default
:rescan–enabled	If t, desktop automatically updates itself by scanning during idle time for changes to icons	t
:rescan–min–idle	Minimum of number of seconds of idle time before desktop rescan	3
:doc–view–max–open–docs	When in "View book as document" mode, number of documents than can be open simultaneously	1
:book–max–open–docs	How many documents can be open simultaneously in a book	4
:desktop–max–open–docs	Maximum number of documents that can be open simultaneously on the desktop	32
:path–limit–length	Maximum number of characters in a pathname	256
:prop–menu	The prop sheet object for the document property menu	
:link–prop–menu	The prop sheet object for the document property menu for linked objects	
:override–clipboard–lock	If t, overrides the lock that prevents two users from accidentally working on the same desktop simultaneously	nil

Lisp data

Effectivity attributes are one way of putting information into a document without having that information show up on the page. But there's another way. You can attach data through Lisp that's visible only through Lisp. And you can have that data saved across sessions (in a comma 6 file for data attached to desktop icons) or saved just during a particular session.

Function 8–36: put–saved–data

(tell *object* mid:put–saved–data *:category data*)

***Returns*: data**

E.g. *(tell (doc–point–cmpn) mid:put–saved–data :owner "mary")*
Returns"mary"

Function 8–37: get–saved–data

(tell *object* mid:get–saved–data *:category*)

***Returns*: data**

E.g. *(tell (doc–point–cmpn) mid:get–saved–data :owner)*
Returns"mary" (if put-saved-data used as in above example)

Function 8–38: put–data

(tell *object* mid:put–data *:category data*)

***Returns*: data**

E.g. *(tell (doc–point–cmpn) mid:put–data :owner "mary")*
Returns"mary"

Function 8–39: get–data

(tell *object* mid:get–data *:category*)

***Returns*: data**

E.g. *(tell (doc–point–cmpn) mid:get–data :owner)*
Returns"mary" (if put-data used as in above example)

These functions allow you to create your own category of data and attach it to any object — not just to components, but to any object Interleaf knows about, including desktop icons, and graphic objects. A category can be anything you like. And the type of data can be any type you want. For example —

(tell object mid:put–data :number–of–times 1)
Returns 1
(tell object mid:get–data :number–of–times 1)
Returns 1
(tell object mid:put–data :users (list "Vic" "Ev" "Carol" "Kathy")
Returns ("Vic" "Ev" "Carol" "Kathy")

These two matching sets of functions are alike except that the *saved* ones save the data across multiple Interleaf sessions; the others maintain the data only during the session.

Chapter 9
Inside the Document

With Interleaf Lisp, you can get at virtually any aspect of a document. In this chapter, we'll explore how you can manipulate components and the text contents of components. In addition, you'll learn how to set document-wide properties such as page size.

Document structure

Interleaf documents can be viewed as a hierarchy of objects: a document that contains components that contain inline components and frames, etc. Frames in turn can contain graphic objects and even their own components (through microdocuments). In addition, Interleaf documents use a master-instance paradigm: existing objects are instances of their masters.

But there are other ways of looking at documents. For example, you could look at a document not as a collection of objects, but as text and graphics laid out on pages.

In fact, Interleaf Lisp has three different views it can take of a document:

- The *structural* view looks at a document in terms of the relation of the various objects in it, such as components and frames. It knows that a particular *subhead* component is followed by a particular *para,* but it doesn't know precisely where on the page those two elements are.
- The *format* view considers the document in terms of the text and graphics on the page. The format view knows where every line ending is and where every object is positioned on every page.
- The *name* view is more abstract. It understands the document primarily in terms of the various masters and instances in it. For example, through the name view, you can look at a master and get at all its instances, even if those instances are scattered hither and yon throughout the document.

From a user's point of view, the *structural* view corresponds to looking only at the component bar (although this over simplifies it because inlines, frames and hidden components

are also part of the structural view). The *format* view corresponds to looking only at what you'd see if you turned the component bar off. And the name *view* in a sense corresponds roughly to what the user "sees" when looking at a component property sheet; in applying a change globally, you are affecting every instance of that component or graphic object (i.e., every object with that name) no matter where it is scattered in the document.

Interleaf Lisp allows you to move through a document using any of these three axes.

Because of the structured nature of Interleaf documents, you can sometimes navigate by moving to the *next* or *previous* object, and sometimes by moving to the *child* or *parent* of an object. For example, for a particular component in the structural view, the *next* would be the next component. But the *child* of it might be an inline component. Similarly, the parent of a component is the document itself.

A word about *tokens* is in order here. Taking a very unstructured view, an Interleaf 5 document consists of a series of tokens. Tokens generally consist of two types of information: the type of token they are and their data. Tokens in a document are typically character tokens which consist of the information that they're a character token and the character itself. When you select text and change its font properties, the system invisibly inserts font tokens that contain that information. Likewise, there are frame and index tokens, among others. We will refer to tokens throughout this chapter even before we get to the section that explains them more fully.

But before we learn how to navigate through a document, we have to learn how to get a document object in the first place.

Getting a document

To work in a document — including getting information about its contents or structure — the document must be open. But "open" doesn't necessarily mean "visible." With I-Lisp, it's possible to open a document in such a way that it doesn't open on screen. In the previous chapter, we looked at the various optional arguments when opening a document:

Function 9–1: open

(tell *desktop–object* mid:open *&args*)

***Returns*: t if object can be opened**

E.g. *(tell (dt–object "mydocument.doc") mid:open :hide–window t)*
Opens document (although invisibly) and returns the document object

But how do you get the open document (visible or not) as an object so you can tell it do things?

Function 9–2: *document*

The I-Lisp variable **document** is always bound to the current document. The current document is, by definition, the document your mouse most recently passed over.

This is a little tricky, however. If you have two documents open and you've been working in one of them, entering text and so forth, and then you move your mouse out of that document and graze the corner of the other document, that other document's header bar may turn white. That document will now be considered to be the current document, even though you haven't altered it at all; you've only brushed it with your mouse. Further, opening a document takes precedence over touching it with a mouse. So, if you've worked in document A, then brushed document B, but then open document C (by hand or through I-Lisp), document C will be the most current document, and the one referenced by the variable **document**.

The way to make a document the most current one through I-Lisp is to tell it to *open*. If it is already open and visible, telling it to open will make its window come to the top and overlay any other open windows. If it isn't visible, then telling it to open has no visible effect, of course.

It is vital to understand that you can *only* work in the current document, whether by hand or through I-Lisp.

For example, say you have two documents, called *ping* and *pong*. Here's how you could open both and work in them.

9–1 work–in–two–documents

```
; Open the first document
(tell (dt–object "ping.doc") mid:open)
; Get it as an object
(setq doc1 *document*)
; Open the second document
(tell (dt–object "pong.doc") mid:open)
; Get it as an object
(setq doc2 *document*)
; Open ping to make it current
(tell doc1 mid:open)
; now we can do something in it
(tell *cmpn–editor* mid:create "para")      ; create a para
; Open pong to make it current
(tell doc2 mid:open)
; Now do something in pong
(tell *cmpn–editor* mid:create "subhead") ; create a subhead
```

Function 9–3: doc–current–icon

(doc–current–icon)

***Returns*: desktop object of the current document**

If you need to get at the current document's icon, *(doc–current–icon)* will do it for you. For example,

9–2 make–ascii–backup

```
(defun make–ascii–backup ()
  (let (current–doc icon new–name)
    ; get current doc's icon
    (setq current–doc (doc–current–icon))
    ; make a copy
    (setq icon (tell current–doc mid:copy (tell current–doc
    mid:get–parent)))
    ; make new name with "bak" in it
    (setq new–name (concat (dt–get–property icon :icon–name) "–bak"))
    ; rename icon
    (dt–set–property icon :icon–name new–name)
    ; insert it into container
    (tell (tell current–doc mid:get–parent) mid:insert icon)
    ; open it hidden
    (tell icon mid:open :hide–window t)
    ; save as ascii
    (tell icon mid:save :ascii)
    ; close it
    (tell icon mid:close)
    ))
```

By the way, *(doc–current–icon)* is the same thing as **document**'s parent:

```
(eql (doc–current–icon) (tell *document* mid:get–parent))
Returns t
```

Navigating

Now that you have a document as an object, how do you get at the objects within the document?

There are a variety of shortcuts which we will discuss in their own sections of this chapter. They are:

Object	Comment
(doc–point–marker)	Marker in the text stream where the visible text caret is
(doc–point–cmpn)	Component or inline that contains the text caret
(doc–point–top–cmpn)	Component that contains the text caret; if the caret is in an inline, the top-level component is returned
(doc–point–line)	Line that contains the caret

Object	Comment
(doc–point–page)	Page that contains the caret
(doc–point–column)	Column that contains the caret

These very handy functions give quick access to the objects around where your text caret is. For example, *(doc–point–cmpn)* returns a component object which you can then tell to do things such as "Change thy fonts!" or "Cut thyself!" (You'll see that the actual object returned has an identification number attached to it. That is, if you evaluate *(doc–point–cmpn)*, something like "#<doc–cmpn 0x2ae46c>" will be returned. This number won't be the same the next time you open the document, so don't count on using it for anything.)

But you don't need to be tied to the text caret. With I-Lisp, you can navigate through an entire document. We'll look at some built-in functions that handle it for you neatly. But first, let's see how you might build such functions by hand so that you'll get a sense of how one navigates through Interleaf documents.

Let's begin by getting the first component of a document. The first component is what you get returned if you ask for the *child* of the current document:

9–3 get–first–cmpn

```
(defun get–first–cmpn ()
  (tell *document* mid:get–child))
```

This is equivalent to telling the document component class *doc–cmpn–class* — which is the logical parent of all components — to give its first child, which you could with the expression *(tell–class doc–cmpn–class mid:get–first)*.

Here is how you can get the last component:

9–4 get–last–cmpn

```
(defun get–last–cmpn ()
  (tell–class doc–cmpn–class mid:get–last))
```

Now let's look through an entire document, component by component. In this case we'll both build a list and put each name into a stickup. The key here is that once we have the first component, we can tell it to give us the next one, and continue until there are none left.

9–5 look–through–doc–toplevel

```
; define a global for the cmpn list
(setq *cmpn–list* nil)
(defun look–through–doc–toplevel ()
; builds global list of all cmpns in a doc
  (let (c)
    ; get first cmpn
    (setq c (tell *document* mid:get–child))
```

```
    ; loop
    (while c
      ; build list
      (push c *cmpn-list*)
      ; put name into stickup, just for demo purposes
      (stk-open (tell c mid:get-name))
      ; get next one
      (setq c (tell c mid:get-next :along :structure)))
))
```

This uses *get–next*:

Function 9–4: get–next

(tell *object* mid:get–next :along :*axis*)

***Returns*: next *object* of that type, if any; else nil**

E.g. *(tell cmpn mid:get–next :along :name)*
Returns next component of the same name

There is another function closely related to *get–next*:

Function 9–5: get–previous

(tell *object* mid:get–previous :along :*axis*)

***Returns*: previous *object* of that type, if any; else nil**

E.g. *(tell cmpn mid:get–previous :along :name)*
Returns previous component of the same name

The default of *get–next* (and *get–previous*) is to look along the *structure* of a document, which means that a component will get the next component from the user's point of view. (Because it is the default, you don't have to specify it if you want to use it, i.e., *(tell cmpn mid:get–next)*.) If you are navigating the document along the *name* axis, however, it would give you the next component with the same name. The order in which you get objects by using *get–next* along the name axis is rather arbitrary; you may well not get the next one in terms of the sequence of pages.

Function 9–5 looks at every top level component. But because it uses *get–next*, it won't find any inline components. For that, we have to use *get–child:*

Function 9–6: get–child

(tell *object* mid:get–child :along :*axis*)

***Returns*: child of *object*, if any; else, nil**

E.g. *(tell cmpn mid:get–child :along :structure)*
Returns child, if any

Here's how you could use this to find all the inlines of a component:

9–6 get–all–cmpn–children

```
(defun get-all-cmpn-children (c)
```

```
; gets all children of a cmpn
(while c
  (do-something c)  ; a made-up function
  (push c *cmpn-list*)
  (if (tell c mid:get-child)
    (get-all-cmpn-children (tell c mid:get-child))) ; recurse
  (setq c (tell c mid:get-next)))
)
```

You invoke this function by executing the following:

```
(setq *cmpn-list* nil)
(get-all-cmpn-children cmpn)
```

This will build a global variable (**cmpn-list**) which contains a list. You could also use it to build a list of all the components and inlines in a document by using the following function that calls *get-all-cmpn-children*:

9-7 get-all-doc-children

```
(defun get-all-doc-children ()
; looks through a doc and builds list of all cmpns
  (setq c (tell *document* mid:get-child))
  (while c
    (if (tell c mid:get-child)
      (get-all-cmpn-children c)
      ; else
      (push  c *cmpn-list*))
  (setq c (tell c mid:get-next)))
)
```

To use this function, you would execute the following:

```
; try it out, making a global
(setq *cmpn-list* nil)
(get-all-doc-children)
```

But these home-brewed techniques are not as reliable or powerful as the ones provided with I-Lisp. Instead of *get-all-doc-children*, you should in fact use:

Function 9-7: doc-scan

(doc-scan *function* &optional :context *context*)

Doc-scan is a powerful function which looks at every object in a document. It sends those objects, one at a time, to a function you've specified. Presumably that function looks at each object and decides whether or not to do something with it. The function is then sent the next object by *doc-scan*, and so forth. (The *context* argument allows you to specify that *doc-*

scan should examine a document other than the current one.) For example, the following routine finds every component whose name has a footnote in it, and sets it to italics.

9–8 footnote–to–italics

```
(defun footnote-to-italics (obj level)
  (if (and (is-of-class obj doc-cmpn-class)
           (string= "footnote" (tell obj mid:get-name))
      (tell obj mid:set-props :italics t))
)

(doc-scan 'footnote-to-italics)
```

Notice that the function called by *doc–scan* takes two arguments. The first is the object being passed to it. The second is a number indicating how far nested the object is. For example, an inline within a toplevel component would have a level number of 1, whereas an inline within an inline would have a level number of 2. This enables you to keep track of the rough structure of the document.

Here's how you could use the *context* argument.

```
; Open the first document
(tell (dt-object "ping.doc") mid:open)
; Get it as an object
(setq doc1 *document*)
; Open the second document
(tell (dt-object "pong.doc") mid:open)
; Get it as an object
(setq doc2 *document*)
; doc2 is open and current, but work in doc1
(doc-scan 'footnote-to-italics :context doc1)
```

There are several variants of *doc–scan*:

Function 9–8: doc–scan–top–level

(doc–scan–top–level *function* &optional :context *context* :level)

E.g. *(doc–scan–top–level 'myfun :context doc1 :level number)*
Looks recursively at every top level component, sending it to function myfun

This function looks only at components, not at inlines or frames or other children of components. If you invoke the function with *:level*, it will invoke itself recursively for each component, so you'll get all the inlines, etc.

Function 9–9: doc–scan–cmpn

(doc–scan–cmpn *function component* &optional :level)

Like our home-brewed *get–all–cmpn–children*, *doc–scan–cmpn* looks at a single specified component and gets the children (inlines and frames). If you specify *:level*, it will also delve into inlines and frames in the component and will get all their components.

Function 9–10: doc–scan–obj

(doc–scan–obj *function object* &optional :level)

***Sends all siblings of* object *to* function**

Doc–scan–obj finds the siblings of any object. Specifying *:level* makes it find all nested children within the object.

Function 9–11: doc–scan–class–apply

(doc–scan–class–apply *function classes*)

E.g. *(doc–scan–class1-apply 'myfun (list dg–named–class))*
Sends every named graphic object to myfun

One of the easy ways to go wrong here is to write a function for use by *doc-scan-class–apply* that has *two* arguments. It only wants one argument — the object that's being examined.

While Lisp mavens may prefer to do raw coding going up and down the chain along the *name* or *structure* access, the *doc-scan* functions make it easy to get a lot done.

Editor objects

There are in any document a set of invisible objects called *editors*. For example, the text editor knows all about how to edit text — cutting, pasting, hyphenating, etc. When the end user makes a popup choice to, say, cut text, he or she is unknowingly addressing his or her command to the text editor. The programmer can direct commands to the editors as well. The programmer can also create a new editor and make it the standard editor for a document or a class of documents. This makes it possible to design unique active documents easily.

The editor objects include

Editor	Controls
doc–editor	Document header pulldowns and other document properties
doc–text–editor	Text popups and other functionality
doc–cmpn–editor	Component bar popups and other functionality
doc–table–editor	Tables popups and other functionality

We will discuss each of these.

The document editors are what gets addressed when a user makes a menu choice. The programmer can directly address those editors, often through a syntax that mimics the standard

popup menu choices. For example, the user can create a frame named, say, "Rule", by choosing **Create ▸Frame ▸Rule** from the popups. The programmer can create a frame by telling the text editor to create a frame named "Rule":

```
(tell *text-editor* mid:create :frame "Rule")
```

Using the editors is a relatively high-level operation. That means that a brief command does a lot. It also means that you get any "side effects" of an action. For example, telling the document editor to close a document brings with it the side effect of getting the close stickup if there have been any changes to the document. Or, changing a font will update the font box in the document header.

We'll begin with the document editor.

Document editor

The document editor for the current document can be addressed through the variable **doc-editor**. This editor is responsible for document-wide actions, most of which the user would initiate through the document's pulldown menus in the document window header.

The document editor has a set of properties that can be accessed through *(tell *doc-editor* mid:get-props)*, optionally using a keyword. Among the properties are:

Property	Explanation
:modify	Amount a document has been modified since it was last saved. Nil or 0 means it hasn't been modified. Every operation adds a number (from 1 to 30). Used to determined if document needs saving before closing.
:page	Current page number. Within a book, gives the proper page number from the beginning of the book. (See text following table for more information.)
:page-range	Cons of first and last page numbers of document. (Cannot be set.)
:view-anchor	Display frame anchors
:view-empty	Display empty text strings in graphics
:view-facing	Display facing pages
:view-index	Display index markers
:view-inline	Display inline markers
:view-return	Display hard returns
:view-ruling	Display invisible table rulings
:view-space	Display spaces as raised dots
:view-tab	Display tab markers
:view-undo	Display inline markers created by the undo function
:zoom	How much the document is zoomed

When you *get–props* for *:page*, you are returned the current page number. When you *set–props*, however, you are taken to a page. For this you must use the keyword *:first*, *:last*, *:next*, or *:previous*. If you give t as an argument, you will be taken to the current page (bringing the top of the page to the top of the window); if you give nil as an argument, you will get a stickup asking you to type in the page number you want to go to. For example,

(tell *doc–editor* mid:set–props :page :last)
Takes you to last page
(tell *doc–editor* mid:set–props :page nil)
Prompts you for number of page to go to

But suppose you want to go to a page not addressed through a keyword, and you don't want to have to prompt the user to type in the page number. You can always open the document to the page you want even if it's already open. For example,

(tell (doc–current–icon) mid:open :page 77)
Takes you to page 77

Function 9–12: close

(tell *doc–editor* mid:close &optional *:force t)

Closes the document; if *:force* is non-nil, closes without asking for confirmation

E.g. *(tell *doc–editor* mid:close :force t)*

If you don't give this method an argument, it will ask you if you want to save, cancel or close (if there have been any modifications). For example,

(tell *doc–editor* mid:close)
If there have been any modifications, prompts user with a stickup before closing

Function 9–13: open–props

(tell *doc–editor* mid:open–props *&optional* :page :color :pattern :printer)

Opens a property sheet for user interaction

E.g. *(tell *doc–editor* mid:open–props :color)*
Opens color property sheet

If no argument is given, then the document property sheet opens.

Function 9–14: save

(tell *doc–editor* mid:save *&keyword* :format :version *&optional* :force)

Saves document

This saves the document in your choice of formats (ASCII or fast) or versions (version 3 for Publisher, 4 for TPS 4 or 5 for Interleaf 5). Whenever you save in version 3 or 4, the document is automatically saved in ASCII format. For example,

(tell *doc–editor* mid:save :format :fast)
Saves in normal fast (binary) format

```
(tell *doc-editor* mid:save :format :ascii)
Saves in ascii format
(tell *doc-editor* mid:save :version 3)
Saves in ASCII version 3 for Publisher
(tell *doc-editor* mid:save :version 4 :format :fast)
Saves in version 4 ASCII; ignores the :format command
(tell *doc-editor* mid:save :format :fast :force t)
Forces a save in normal fast (binary) format, even if there have been no
modifications
```

Function 9–15: convert

(tell *doc-editor* mid:convert *&key* :version *number* *&optional* :name *name*)

Makes a new copy of document, under *name*, converted to version *number*

E.g. *(tell *doc-editor* mid:convert :version 4 :name "mydoc4")*
Creates copy named "mydoc4.doc" saved in version 4

Unlike specifying a version number (3, 4 or 5) with *mid:save*, *convert* creates a new copy. If you don't specify a name, the system will prompt the user to supply one.

Function 9–16: rename

(tell *doc-editor* mid:rename *&optional name*)

Renames file to *name*

E.g. *(tell *doc-editor* mid:rename :name "text1")*
Creates new copy with name "text1"

This does exactly what *rename* does when a user chooses it from the document header menu: it creates a new copy under the new name, closes the original copy, and leaves the user in the new copy.

If you don't supply it with a name, it will prompt the user for one. If you do supply a name, it makes a copy with that name.

If you want to make a copy but not have to open the document first to do it, you can use the *copy* command on the desktop. For example,

```
; get the selected object
(setq doc (dt-child-selected))
; make a copy of it, put it on the desktop, and get the copy
(setq newdoc (tell doc mid:copy *dt-desktop*))
; rename the new object
(dt-set-property newdoc :icon-name "New name")
```

Function 9–17: revert

(tell *doc-editor* mid:revert *&key* :which :document)

Reverts to previous version of document

There are five possible arguments to *:which*— *:document*, *:backup*, *:checkpoint*, *:crash* or *:work*. These all correspond to choices the user sees on the revert pulldown ("*:work*" refers to the work-in-progress file).

If you specify *:force* with a non-nil value, it will do the reversion without first querying the user.

(tell *doc-editor* mid:revert :which :backup :force t)
Reverts to backup without asking for confirmation

Document manipulation

You can also manipulate documents without going through the document editor simply by telling the document to do different things.

You can get and set properties of document objects. (RSUs, or "ridiculously small units," are Interleaf's internal unit of measurement. There are 1,228,800 rsu's in an inch.)

Function 9–18: document get–props

(tell *document* mid:get–props *&optional keyword*)

***Returns*: property requested through keyword, or all properties if no keyword**

E.g. *(tell *document* mid:get–props :columns)*
Returns number of columns

Function 9–19: document set–props

(tell *document* mid:set–props *&optional keyword value*)

Sets *document*'s property of *keyword* to *value*

E.g. *(tell *document* mid:set–props :columns 2)*
Makes document into a two–column document

There are many properties you can get and set. The following table lists most of them. It is organized along the various property sheets that provide end-user access to these properties. (If you need more information about what these properties do, check your Interleaf 5 documentation.)

	Page Property Value
Basic Property	
:columns	Number of columns
:gutter–width	In rsu's
:balance–columns	t or nil
:page–width	In rsu's
:page–height	In rsu's
:top–margin	Top margin of page, in rsu's
:bottom–margin	Bottom margin of page, in rsu's

:left–margin	Left margin of page, in rsu's
:right–margin	Right margin of page, in rsu's
:inner–margin	Inner margin of page, in rsu's
:outer–margin	Outer margin of page, in rsu's
:page–layout	Choice of *:single–sided*, *:odd–pages–right*, *:odd–pages–left*. If *:single–sided*, inner and outer margins are ignored.
:turn–layout	Choice of *:normal*, *:clockwise*, *:counter–clockwise*
:header–footer–bleed	If t, headers and footers stretch all the way across the page
:different–first–header	t or nil
:different–first–footer	t or nil
:first–header	Frame used for first header. Cannot be set
:first–footer	Frame used for first footer. Cannot be set
:right–header	Frame used for right header. Cannot be set
:left–header	Frame used for left header. Cannot be set
:right–footer	Frame used for right footer. Cannot be set
:left–footer	Frame used for left footer. Cannot be set
Custom Property	
:general–unit	Preferred unit of measurement. Choice of *:inches*, *:points*, *:mm*, *:picas*, *:ciceros*, *:didots*
:line–spacing–unit	Preferred unit of measurement for line spacing. Choice of *:inches*, *:points*, *:mm*, *:picas*, *:ciceros*, *:didots*
:font–unit	Preferred unit of measurement for fonts. Choice of *:inches*, *:points*, *:mm*, *:picas*, *:ciceros*, *:didots*
:ascii–unit	Preferred unit of measurement when saving in ASCII. 0 means inches, 1 means mm
:hyphenate	t if hyphenation is on, nil if it's off
:ladder–count	Integer (0–4) specifies number of consequetive lines that end with hyphens. 0 means any number are permitted.
:allow–break–after–hyphen	If t, page or column can end with a hyphen
:cmpn–margin–method	Choice of *:add*, *:minimum*, *:maximum*, *:bottom*, *:top*. Determines relationship of bottom of a margin with top of the next one.
:baseline–to–baseline–margins	t or nil
:rev–bar–placement	Choice of *:automatic*, *:right*, *:left*
:vertical–justification	t or nil
:cmpn–margin–stretch	100% = 1024
:cmpn–margin–shrink	100% = 1024
:frame–margin–stretch	100% = 1024
:feathering	t or nil
:justify–pages	t or nil

:long–page–justify–threshold	100% = 1024
:short–page–justify–threshold	100% = 1024
:autonumbers–frozen	t or nil
:composition–frozen	t or nil

Printer Property	Value
:header–page	If t, add header page when printing
:double–sided	If t, print double sided
:manual–sheet–feed	If t, feed paper manually
:print–rev–bars	If t, print rev bars
:print–strikes	If t, print strike-throughs
:print–underlines	If t, print underlines
:print–deletion–marks	If t, print deletion marks used by rev tracking
:underline–position	Choice of *:baseline*, or *:descender*
:default–printer	String indicating default printer
:orient–same–as–page	If t, printing matches page orienation
:spot–color–separation	Choice of *:solid*, *:screened*, *:off*

There are other properties you can get and/or set.

Miscellaneous	Value
:window	Window object for the document. Cannot be set
:keymap	The keymap for this document (see chapter on keyboard)
:read–only	If t, document is read-only
:color–palette	Document's color palette
:pattern–palette	Document's pattern palette
:doc–type	1 if binary, 2 if ASCII markup, 3 if plain ASCII. Cannot be set.

In addition to setting and getting properties, you can set various variables that affect the document.

Function 9–20: doc–get–vars

(doc–get–vars *&optional keywords*)

***Returns*: list of settings**

E.g. *(doc–get–vars :initial–zoom)*
Returns 1.0

Function 9–21: doc–set–vars

(doc–set–vars *keyword value*)

Sets variable *keyword* to *value*

E.g. *(doc-set-vars :initial-zoom 1.8)*
Sets initial zoom variable to 1.8

Here is a list of some of the variables you can set.

:initial-zoom	Amount of zoom when the document opens
:advanced-formatter	If t, uses different algorithms for formatting (e.g., figuring line breaks), based on Knuth/Plass. Interleaf, Inc. does not officially support this option.
:checkpoint-control	Integer from -100 to 100. Controls how frequently checkpoint copies are made. If 0, no checkpointing.
:allow-break-after-hyphen	If t, "allow break after" shows up on page property sheet

You can get further information about the document window from four related functions.

Function 9-22: doc-pixel-window-top-offset

(doc-pixel-window-top-offset)

***Returns*: offset of document window from top of screen, in screen pixels**

E.g. *(doc-pixel-window-top-offset)*
Returns 58

Function 9-23: doc-pixel-window-top-offset

(doc-pixel-window-left-offset)

***Returns*: offset of document window from the left of screen, in screen pixels**

These amounts inlcude the size of the component bar and document header. They measure the distance of the actual screen representation of the upper left corner of the page (the white area) from the edge of the screen.

Function 9-24: doc-pixel-window-height

(doc-pixel-window-height)

***Returns*: height of current document's window, in screen pixels**

Function 9-25: doc-pixel-window-width

(doc-pixel-window-width)

***Returns*: width of current document's window, in screen pixels**

Markers

Before moving on to the text editor, we're going to talk about markers. Markers are invisible pointers into the document's text contents. Markers should not be confused with the visible marker — the text caret. The text caret is, in fact, a type of marker, but not all markers are visible.

Function 9–26: doc–point–marker
(doc–point–marker)
***Returns*: a marker at the position of the text caret**

(doc-point-marker) gives you a marker that points to wherever the text caret is currently pointing.

```
(setq mark (doc–point–marker))
Returns marker; sets mark equal to that marker
```

Function 9–27: get–marker
(tell *component* mid:get–marker *nil or t*)
Returns*: marker at beginning or end of *component

E.g. *(tell (doc–point–cmpn) mid:get–marker nil)*
Returns marker at beginning of component

If you give *get–marker* nil as an argument, it returns a marker at the very beginning of the component (even before the font token, as we will discuss in the section on tokens); if you give it as an argument, it gives you a marker at the end of the component. If you don't give it any argument, it defaults to nil.

Function 9–28: marker copy
(tell *marker* mid:copy)
Returns*: copy of *marker

E.g. *(setq m2 (tell m1 mid:copy)*
Returns new marker at position of m1

One reason this function is useful is that it allows you to get a marker, make a copy, move the copy some fixed amount, and now use the original and the copy as delimiters marking a text string.

Function 9–29: doc–text–selection
(doc–text–selection)
***Returns*: if text is selected, returns dotted pair of markers; else, nil**

This function gives you markers at the beginning and end of selected text.

Function 9–30: doc–goto–marker
(doc–goto–marker *marker*)
Returns*: moves text caret to *marker

This function moves the text caret to the marker passed as a parameter. If that marker is invalid or for some reason the text caret can't be placed there, nil is returned. Among the reasons why the text caret can fail to be placed at a marker: the component with the marker was cut, the component is invisible due to effectivity, or the marker is in a microdocument that can't be edited.

The following is a fairly useless bit of code that illustrates some of these marker functions. If there is text selected, it moves the text caret to either end of it 100 times. Otherwise, it tells you that no text is selected.

9–9 move–caret–in–selected–text

```
(defun move–caret–in–selected–text ()
  (let (markers mark (ctr 1))
    ; is any text selected?
    (if (setq markers (doc–text–selection))
      (progn
        ; Begin loop
        (repeat 100
          ; alternate which marker you get
          (if (oddp (inc ctr))
            (setq mark (car markers))
            ; else
            (setq mark (cdr markers)))
          (doc–goto–marker mark)
          ; force the change to be visible
          (doc–flush–queue)))
      ; else nothing selected
      (stk–open "Nothing is selected."))
))
```

If you actually try this, you'll see that it not only works, it's really absolutely useless.

But don't worry — there are plenty of useful things we can do with markers.

Moving Markers

By moving a marker, you can select text to affect, and can look at the characters along the way. Remember that when you are moving a marker through a component, you can tell if you've reached the end if the return is greater than 0, since the marker *move–by* method returns the number of tokens it tried and failed to move by.

Function 9–31: move–by

(tell *marker* mid:move–by *count &optional :by &keywords :word–endings :word–beginnings :component–beginnings :component–endings)*

***Returns*: how many left after hitting the end of the cmpn or doc**

E.g. *(tell marker mid:move–by 3 :by :word–endings)*

This will move the marker the number of tokens specified by *count*. If you specify a negative count, you move backwards. If you hit the edge of the component before you've used up your count, the function returns the number left in the count.

This method can take parameters. If you add *:by* followed by what you want to move by, you can move from word to word or component to component. The keywords are: *:word–beginnings*, *:word–endings*, *:component–beginnings,* and *:component–endings.*

Function 9–32: move–to

(tell *marker* mid:move–to *count &optional :by &keywords :word–endings :word–beginnings :component–beginnings :component–endings)*

***Returns*: how many left after hitting the end of the cmpn or doc**

E.g. *(tell marker mid:move–to 3 :by :word–endings)*

This function is very similar to *(tell* marker *mid:move–by* count) except that it counts from an absolute position at the beginning of the component.

Here are three related functions to count tokens in a component, words in a component, and components in a document.

9–10 count–tokens–in–cmpn

```
(defun count–tokens–in–cmpn (cmpn)
  (let (m (ctr 0))
    ; get marker at beginning of cmpn
    (setq m (tell cmpn mid:get–marker))
    ; loop until no tokens left
    (while (zerop (tell m mid:move–by 1))
      ; increment counter
      (inc ctr))
  ; return the ctr
  ctr
  ))
```

9–11 count–words–in–cmpn

```
(defun count–words–in–cmpn (cmpn)
  (let (m (ctr 0))
    ; get marker at beginning of cmpn
    (setq m (tell cmpn mid:get–marker))
    ; loop until no words left
    (while (zerop (tell m mid:move–by 1 :by :word–endings))
      ; increment counter
      (inc ctr))
  ; return the ctr
  ctr
  ))
```

9–12 count–cmpns–in–doc

```
(defun count–cmpns–in–doc (cmpn)
  (let (m (ctr 0))
    ; get marker at beginning of cmpn
    (setq m (tell cmpn mid:get–marker))
    ; loop until no words left
```

```
    (while (zerop (tell m mid:move-by 1 :by :cmpn-endings))
      ; increment counter
      (inc ctr))
  ; return the ctr
  ctr
  ))
```

You use markers to select text. But to do so, you have to tell the document's text editor to perform some commands. The text editor is the editor object responsible for enabling you to edit text within an Interleaf document. We will discuss it further later.

Function 9–33: select

(tell *text-editor* mid:select *marker1 marker2)*

***Returns*: t if successful; else nil**

This selects the text between *marker1* and *marker2*.

Function 9–34: deselect

(tell *text-editor* mid:deselect)

***Returns*: t if successful; else nil**

This deselects the text between *marker1* and *marker2*.

Here are some examples of using the selection commands:

9–13 flicker-selection

```
(defun flicker-selection ()
; flickers selected text; some machines don't display this well
  (let (markers m1 m2)
    ; need initial selection
    (if (setq markers (doc-text-selection))
      (progn
        ; get the markers
        (setq m1 (car markers))
        (setq m2 (cdr markers))
        ;loop 10 times
        (repeat 10    ; faster machines might want to make this 100
          (tell *text-editor* mid:deselect)
          (tell *text-editor* mid:select m1 m2))))
))
```

9–14 select-a-word

```
(defun select-a-word (marker &optional backwards)
; Selects word starting at marker. If backwards is non-nil, selects
backwards
  (let (m2 direction remaining)
```

```
    ; go backwards?
    (if backwards
      (setq direction -1)
      ; else
      (setq direction 1))
    ; deselect previous selection, if any
    (tell *text-editor* mid:deselect)
    ; make copy of marker
    (setq m2 (tell marker mid:copy))
    ; move copy to end of word
    (setq remaining (tell m2 mid:move-by direction :by :word-endings))
    ; select the word
    (tell *text-editor* mid:select marker m2)
    ; If move was possible, return m2, else return nil
    (if (> remaining 0)
      (setq m2 nil))
m2
))
```

9–15 select–each–word

```
(defun select-each-word (cmpn)
; selects each word in a component, in turn
  (let (m1)
    ; get marker at beginning of cmpn
    (setq m1 (tell cmpn mid:get-marker nil))
    ; loop through component until end
    (while m1
      (doc-flush-queue) ; force an update
      (setq m1 (select-a-word m1))
      ; italicize, just for demo
      (tell *text-editor* mid:set-props :italic t))
    (tell *text-editor* mid:deselect)
))
```

9–16 italicize–line

```
(defun italicize-line (line)
  (let (m1 m2)
    ; move to beginning of line
    (key-begin-line)
    ; get the marker
    (setq m1 (doc-point-marker))
    ; move to end of line
    (key-end-line)
```

```
    ; get the marker
    (setq m2 (doc-point-marker))
    ; select the line
    (tell *text-editor* mid:select m1 m2)
    ; italicize line
    (tell *text-editor* mid:set-props :italic t)
    ; deselect the line
    (tell *text-editor* mid:deselect)
))
```

This function uses commands normally attached to keystrokes — *key-begin-line* and *key-end-line* — to move to the beginning and end of the line. There we grab markers, use them to select the line of text, and then use the text editor to italicize the line. (Later you'll learn how to set markers at the beginning and end of the line without actually forcing the text caret to change its position.) Of course, for Interleaf 5, a line is an ephemeral thing; entering or deleting a character elsewhere in the component may change what constitutes the line. In that case, the string of text will remain italicized, even if it no longer constitutes a line. But this function is useful when you are using hard returns to indicate new lines. For example, you might use this to italicize the comments in some programming code.

Of course, you could modify this function to perform other font operations, or even to delete lines. See the section on the text editor for more information.

Here are some more functions associated with markers.

Function 9–35: marker get-parent

(tell *marker* mid:get-parent &optional :along :structure :format)

***Returns*: containing component (if along structure), containing line (if along format)**

Taken just the way it is, without any further arguments, this will give you the component or inline in which the marker is located. That's because without any further arguments, this function defaults to *:along :structure*. If, however, you specify *:along :format*, you will get the line in which the marker is located. For example:

```
(tell (doc-point-marker) mid:get-parent)
Returns component marker is currently in
(tell (doc-point-marker) mid:get-parent :along :format)
Returns line object
```

The following function, for example, will tell you what component your marker is in and the width of the line:

9–17 where-and-how-long

```
(defun where-and-how-long ()
  (let (line line-in-inches (marker (doc-point-marker)))
    ; get cmpn parent of marker
```

```
    (setq cmpn (tell marker mid:get-parent :along :structure))
    ; get line as parent of marker
    (setq line (tell marker mid:get-parent :along :format))
    ; get width of line and convert to inches (from RSU's)
    (setq line-in-inches
      (/ (tell line mid:get-props :width) 1228800.0))
    ; display info in a stickup
    (stk-open
      (format nil "In component: ~A\nWidth: ~D" (tell cmpn mid:get-name)
        line-in-inches))
))
```

Function 9–36: get-top-cmpn

(tell *marker* mid:get-top-cmpn)

***Returns*: top level containing component**

This returns the component that contains the marker. If the marker is in an inline, it returns the top-level component. This can be very handy if, for example, you are constructing a form that uses inlines. Suppose you have a couple of inlines used repeatedly throughout the form because they contain buttons or do range checking. But you may need to know what component they're in to see how they should act in this or that circumstance. This function allows you to get at that top-level component.

Function 9–37: get-bounds

(tell *marker* mid:get-bounds)

***Returns*: position of marker, relative to page, as a dotted pair of pixels**

E.g. *(tell (doc-point-marker) mid:get-bounds)*
Returns (117 . 410)

This function allows you to link a position within a component to its position on a page, for it tells you the horizontal and vertical position of the marker on screen. It uses the pixel as the unit of measurement. The numbers it gives are relative to the page, which means they stay the same even if you move the document window around or resize it.

Here's one use of this:

9–18 set-initial-indent

```
(defun set-initial-indent ()
; sets initial indent to caret position. Can't decrease initial indent.
  (let (pixelpos pixelx indent innermarg)
    ; get current marker position
    (setq pixelpos (tell (doc-point-marker) mid:get-bounds))
    ; get x coordinate
    (setq pixelx (car pixelpos))
    ; figure indent, converting pixels to RSU's
```

```
    ; (figure 75 pixels per inch, and 1228800 RSU's per inch)
    (setq indent (* (/ pixelx 75.0) 1228800))
    ; get left page margin offset
    (setq innermarg
      (tell *document* mid:get-props :inner-margin))
    ;(setq innermarg (/ innermarg 1228800.0))
    ; figure indent (marker position on pg minus inner pg margin)
    (setq indent (round (- indent innermarg)))
    ; set the indent
    (tell (doc-point-cmpn) mid:set-props :initial-indent indent)
))
; try it, binding it to ^Xy
(kbd-bind kbd-doc-map "\^Xy" 'set-initial-indent)
```

Function 9–38: get–location

(tell *marker* mid:get–location)

Returns*: dotted pair of *component number* and *token–number

E.g. *(tell (doc–point–marker) mid:get–location)*
Returns (196 . 9) (i.e., 196th component, 9th token)

Interleaf numbers components starting from zero; the first component is component number 0. Also, notice that *get–location* counts all components hidden by effectivity.

Function 9–39: is–in–word

(tell *marker* mid:is–in–word)

***Returns*: t if marker is in a word; else nil**

But what counts as a word? Clearly, if the marker is in the middle of a word, this function will return t. It will also return t if the marker is between a space and the first letter of a word. But it returns nil if it's between the last letter and a space. It can be hard to predict exactly what will count as being within a word, as the following illustration shows.

Function 9–40: insert

(tell *marker* mid:insert string *or token–list*)

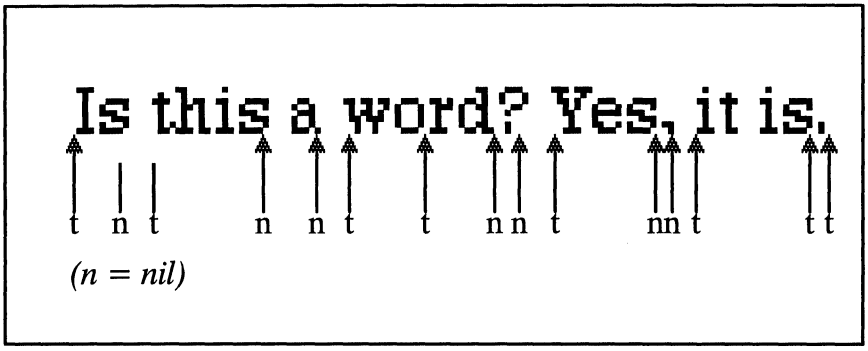
***Returns*: t unless *marker* is in read-only component**

E.g. *(tell (doc–point–marker) mid:insert "Hello")*
Returns t and inserts "Hello" at component

The following inserts at the text caret whatever text one has typed into a stickup.

9–19 insert–string–at–caret

```
(defun insert-string-at-caret ()
  (let (text)
    (setq text (stk-open "Enter text to be entered: " :input 60))
    (tell (doc-point-marker) mid:insert text)))
```



Markers also provide a way for you to insert and delete text and other objects.

The next function inserts your name at the text caret.

9–20 insert–name

```
(defun insert–name ()
; inserts name as defined in variable in profile.drw
  (tell (doc–point–marker) mid:insert *my–name*)
  )
```

This function assumes you have put the line *(setq *my–name* "Harriet Higgins")* (or whatever) in your profile drawer. You'd probably want to bind this function to a keystroke to have it actually be useful.

The following two functions expand on this, and use *doc–scan* to insert the day's date into any component named *date*. Note that it first deletes the current contents of the *date* component.

9–21 auto–insert–date

```
(defun auto–insert–date ()
; build a string expressing the date
(let (date–str tm m m2 name)
  ; get the time ... a list of milliseconds, seconds, etc.
  (setq tm  (get–decoded–time))
  ; use format to make a string
  (setq date–str (format nil  "~A ~D, ~4,'0D"
    (nth (nth 4 tm)
    (list nil "January" "February" "March" "April" "May"
      "June" "July" "August" "September" "October"
      "November" "December"))
    (nth 3 tm) ; day
    (nth 5 tm) ; year
  ))
```

```
; Look through document for cmpns named "date"
; (uses lambda, which we don't explain)
 (doc-scan '(lambda (o l)
  (if (and (setq name (advise o mid:get-name))
    (string= "date" name))
     ; if we've found a date cmpn
      (progn
        (setq m (tell o mid:get-marker))
        (setq m2 (tell o mid:get-marker t))
        ; delete the current contents
        (tell m mid:delete m2)
        ; insert the date
      (tell m mid:insert date-str)))))
      ))
;; bind it to a keystroke -- ^Xd
(kbd-bind kbd-doc-map "\^Xd" '(auto-insert-date) )
```

The following function inserts a word count in brackets every tenth word.

9–22 insert–incremental–word–count

```
(defun insert-incremental-word-count (cmpn interval)
  (let (count marker number-string)
    ; Initialize count
    (setq count 0)
    ; get marker at beginning of cmpn
    (setq marker (tell cmpn mi:get-marker))
    ; move ahead a word at a time
    (while (zerop (tell marker mid:move-by 1 :by :word-endings))
      ; increment counter
      (inc count)
      ; check for zero remainder when divided by interval
      (if (eql 0 (% count interval))
        (progn
          ; create number string to insert
          (setq number-string (concat "<" (itoa count) ">"))
          ; insert number
          (tell marker mid:set-props :follow-insert t)
          (tell marker mid:insert number-string))))))
```

By using the *(tell* marker *mid:move–by* count *:by :cmpn)* in the above function, you could alter it to insert a number before every paragraph. (You'd be wiser to insert an auto-number, however).

The *(tell* marker *mid:insert)* function allows you to insert a plain text string or other tokens. If you want to insert tokens, you have to construct a dotted pair list.

Function 9–41: delete
(tell *marker1* mid:delete *marker2*)
***Returns*: t if successful; else, nil**
E.g. *(tell m1 mid:delete m2)*

The above function lets you delete tokens between two markers.

The following function deletes all of a component's contents:

9–23 delete–cmpn–contents

```
(defun delete–cmpn–contents (cmpn)
  (let (m1 m2)
    ; get marker at beginning of cmpn
    (setq m1 (tell cmpn mid:get–marker nil))
    ; get marker at end of cmpn
    (setq m2 (tell cmpn mid:get–marker t))
    ; delete it
    (tell m1 mid:delete m2)
))
```

This is a very useful function whenever you want to update a component by erasing its current contents and replacing them with new contents.

You should find it straightforward to adapt the *italicize–line* function we created above to delete a line.

Here are a set of functions that will alphabetize the words in a component.

9–24 cmpn–into–wordlist

```
(defun cmpn–into–wordlist (c)
; returns a list of words in cmpn c
(let (m1 m2 (word–list nil))
  (setq m1 (tell c mid:get–marker ))  ; get marker
  (setq m2 (tell m1 mid:copy))        ; make a copy of it
  ; loop, getting each word
  (while (= 0 (tell m1 mid:move–by 1 :by :word–beginnings))
    (tell m2 mid:move–by 1 :by :word–endings)
    ; get word, using get–substring not discussed yet
    ; variable word is defined in alphabetize–component
    (setq word (tell m1 mid:get–substring m2 t t))
    (if word (push word word–list))
)
word–list
))
```

9–25 word–sort

```
(defun word–sort (a b)
; compare routine for alphabetizing
  (> (string–compare b a) 0))
```

9–26 alphabetize–component

```
(defun alphabetize–component (cmpn)
; alphabetizes contents of a cmpn
; strips out frames, tokens, etc.
  (let (wordlist word m)
    (setq m (tell cmpn mid:get–marker))
    (tell m mid:set–props :follow–insert t)
    ; get the wordlist
    (setq wordlist (cmpn–into–wordlist cmpn))
    ; sort it using our own word–sort function
    (setq wordlist (sort wordlist 'word–sort))
    ; now, if you want, delete contents of cmpn,
    ; using delete–cmpn–contents 9–9–23
    (delete–cmpn–contents cmpn)
    ; insert words by looping through wordlist
    ; first, get marker at end of empty cmpn
    ; (if at beginning, you'd have to
    ;  to move past font marker)
    (setq m (tell cmpn mid:get–marker t))
    (while (setq word (pop wordlist))
      (tell m mid:insert  (concat word "\n"))
      (setq m (tell cmpn mid:get–marker t)))
    (doc–flush–queue)
  ))
```

Notice that these functions sort along ASCII order. And whether they are case–sensitive depends on how the **case-sensitive** variable is set; this is a system variable that causes *string–contained* (among other functions) to be case–sensitive when it is set to t, and not when it is set to nil.

Also, these functions are not very sophisticated. They don't handle components that have frames or index tokens, and they are dumb about handling changes in font and the like. And if a word is used twice or more in a component, it gets listed twice or more. You might consider enhancing these functions as homework.

Tokens

We have several times referred to tokens while promising we would explain them later. Now is the time.

Tokens are objects encountered in the stream of characters in an Interleaf component. For example, a line of text might include a set of character tokens (i.e., the characters that show up on screen), a font-change token, tokens indicating hard or soft spaces, and so forth. A token consists of two basic pieces of information, generally its type and its content. For example, a character token consists of a dotted pair whose car is *:char* and whose cdr is the identifying number of that character.

Token	Description	Example
:char	A character such as "0–9", "A–Z", "a–z"	(:char . #\A)
:hyph–char	A character after which a hyphen can be inserted. When the spell checker detects a misspelled word, it flags it by making its last character a hyph-char	(:hyph–char . #\e)
:doc–font	Font change information (or, if first token in component, that component's base font)	(:doc–font . #<doc–font 0x8f64a0)
:hard–return	What you get when you hit the Return key	(:hard–return)
:inline–head	Indicates the beginning of an inline component	(:inline–head . #<doc–cmpn 0x8678B0>)
:inline–tail	Indicates the end of an inline component	(:inline–tail . #<doc–cmpn 0x8678B0>)
:frame	Frame	(:frame . #<doc–frame 0xcebc6c>)
:index	An index token — not to be confused with the visible index tokens. Even if you hide the visible ones, this index token remains to tell the system there's indexing to be done here	(:index #<doc–index–mark 0xd27800>)
:glue	Spacer between words to justify a line, expressed in screen pixels	(:glue .4)
:tab	Tab stop. Lists type of tab (:blank, :dotted, :underline :center–line), tab-stop number, and width in screen pixels	(:tab :blank 1 75)
:autonumber	Autonumber	(:autonumber . #<doc–autonum–mark 0xa2f70e>)
:cross-reference	Cross-reference to an autonumber	(:cross–reference . *cross-reference-props*)

We will look at these in some detail. But first we need a function that will return a token. And for this, we need the more general–purpose function, *get-substring*

Function 9–42: get–substring

(tell *marker1* mid:get–substring *marker2* &optional *token–types in–string*)

Returns*: tokens between *marker1* and *marker2

E.g. *(tell m1 mid:get–substring m2 (list :char :hyph–char) t)*
Returns string of all characters and hyphenated characters between m1 and m2

This is a powerful function that will give you all of the tokens between two markers in either list or string format. You can also give it a list of the types of tokens you want. For example, suppose you put the text caret into a component that consists of the words "Hello, friend!" followed by a frame:

```
(setq m1 (tell (doc–point–cmpn) mid:get–marker nil))
Returns marker at beginning of current cmpn
(setq m2 (tell (doc–point–cmpn) mid:get–marker t))
Returns marker at end of current cmpn
; return it as string
(tell m1 mid:get–substring m2 t t)
Returns "Hello, friend!"
; return it as list of all tokens
(tell m1 mid:get–substring m2 t)
Returns ((:doc–font . #<doc–font 0x8d98c0>) (:char . #\H) (:char . #\e)
(:hyph–char . #\l) (:char . #\l) (:char . #\o) (:char . #\,) (:glue . 4) (:char .
#\f) (:char . #\r) (:char . #\i) (:char . #\e) (:char . #\n) (:char . #\d) (:char .
#\!) (:frame . #<doc–frame 0xa66110>))
; return only frames
(tell m1 mid:get–substring m2 (list :frame))
Returns ((:frame . #<doc–frame 0xa66110>))
; return string of characters — no hyphenated characters (such as the first
"l"), no spaces ("glue")
(tell m1 mid:get–substring m2 (list :char) t)
Returns "Helo,friend!"
; return string of characters , hyphenated characters and spaces ("glue")
(tell m1 mid:get–substring m2 (list :char :hyph–char :glue) t)
Returns "Hello, friend!"
```

We can use *get–substring* to give us a selected portion of a component. For example:

9–27 selected–text–substring

```
(defun selected–text–substring ()
; returns substring of selected text
  (let (m1 m2 substr)
    ; initialize substring
    (setq substr nil)
    ; if any text is selected
```

```
    (if (doc-text-selection)
      (progn
        ; get markers at beginning and end of selection
        (setq m1 (car (doc-text-selection)))
        (setq m2 (cdr (doc-text-selection)))
        ; get the substring
        (setq substr (tell m1 mid:get-substring m2))
      )
      ; else, nothing selected
      (stk-open "No text is selected"))
    ; return selected text as a list
    substr
))
```

9–28 get-token-substring

```
(defun get-token-substring (m1)
; returns the token immediately following marker m1
  (let (m2)
    ; make a copy of m1
    (setq m2 (tell m1 mid:copy))
    ; move the copy one token forward
    (tell m2 mid:move-by 1)
    ; get the substring and return its first and only element
    (car (tell m1 mid:get-substring m2))
))
```

Text editor

The text editor is an invisible object that knows how to do things with text. Users access the text editor primarily through popups and pulldowns when in the text area of a document. The Interleaf Lisp developer can access the text editor more directly.

The text editor for the current document can always be reached through the variable **text-editor**.

The text editor has a set of properties that can be accessed through *(tell *text-editor* mid:get-props)*, optionally using a keyword. The following are among the properties.

Property	Value
:bold	t, nil or :toggle
:caps	:all-caps, :all-small-caps, :caps-and-small, or nil
:dbl-underline	t, nil or :toggle
:doc-font	The current font
Property	Value

:family	Font family
:italic	t, nil or :toggle
:last–font	Previous font used
:overbar	t, nil or :toggle
:pair–kerning	t, nil or :toggle
:revision–bar	t, nil or :toggle
:size	Font size in points
:strikethrough	t, nil or :toggle
:subscript	t, nil or :toggle
:superscript	t, nil or :toggle
:tab–type	:blank, :dotted, :underscore, :centerline, :dashed
:track–kerning	:tighter, :looser, :tight 1–3, :loose 1–3
:underline	t, nil or :toggle

As you can see, these reflect the choices in the text prop pulldown in the document window header bar.

If you use *set–props* to set a property of the text editor, if there is any selected text, that text will take on the new properties. If there is no selected text, then anything typed at the insertion point will take on the new properties; this too mirrors how the user normally interacts with the text editor.

The following will take a number of characters, starting at the text caret, and increase each one's font size in one–point increments, so it looks like this: example.

9–29 font–mount

```
(defun font–mount (number–of–chars)
  (let (m1 m2 token font–size)
    ; get the starting point
    (setq m1 (doc–point–marker))
    ; get the initial font size
    (setq font–size (tell *text–editor* mid:get–props :size))
    ; loop
    (repeat number–of–chars
      ; get a character by getting  two markers
      (setq m2 (tell m1 mid:copy))
      (tell m1 mid:move–by 1)
      (setq token (tell m2 mid:get–substring m1))
      ; is token a char or hyphenated char?
      (if (member (car(car token)) '(:char :hyph–char) 'equal)
        (progn
          ; increase the font size
          (inc font–size)
```

```
        ; select the text
        (tell *text-editor* mid:select m2 m1)
        ; boost its size
        (tell *text-editor* mid:set-props :size font-size)))
    )
))
```

Here's another stupid font trick.

9–30 bold-flicker

```
(defun bold-flicker ()
; flickers selected text between bold and italics
  (let (m1 m2 original-font)
    ; save original font
    (setq original-font (tell *text-editor* mid:get-props :doc-font))
    (if (not (doc-text-selection))
      (progn
        (stk-open "No text selected")
        (quit)))
    (setq m1 (car (doc-text-selection)))
    (setq m2 (cdr (doc-text-selection)))
    (repeat 7
      (tell *text-editor* mid:select m1 m2)
      (tell *text-editor* mid:set-props :bold :toggle)
      (tell *text-editor* mid:set-props :italic :toggle)
      ; just for effect, flash the selection also
      (tell *text-editor* mid:deselect)
    )
    ; restore it to its original good looks
    (tell *text-editor* mid:select m1 m2)
    (tell *text-editor* mid:set-props :doc-font original-font)
))
```

Remember, *(doc-text-selection)* is a shortcut for *(tell *text-editor* mid:get-selection)*.

Function 9–43: get-selection

(tell *text-editor* mid:get-selection)

***Returns*: Cons of markers at beginning and end of selection**

E.g. *(tell *text-editor* mid:get-selection)*
Returns (#<doc-marker 0x132e05> . #<doc-marker 0x132ef6>

Function 9–44: open-props

(tell *text-editor* mid:open-props)

Does *props* on selected text

When the user selects text and chooses *props* from the popup menu, the system puts information about the selected text into the status bar. If the user has selected an inline component marker, the system opens the inline property sheet. The same things happen when the developer uses the *open–props* function.

Function 9–45: copy

(tell *text–editor* mid:copy)

Puts a copy of the selected text on to the clipboard

This function only works if text is selected, in which case it copies it to the clipboard, which means it's ready to be pasted.

Function 9–46: cut

(tell *text–editor* mid:cut)

Deletes selected text and puts it on to the clipboard

This function only works if text is selected, in which case it cuts the selected material and copies it to the clipboard, which means its ready to be pasted.

If you want to cut material without first having to select it (which means that it gets highlighted on screen before it is cut), you can use the marker method of deleting:

```
(tell marker1 mid:delete marker2)
```

This, however, does *not* put the deleted material on to the clipboard, so you cannot then paste it.

Function 9–47: paste

(tell *text–editor* mid:paste)

Pastes from clipboard into document at text caret

This function works only if no text is selected. In addition, there has to be something on the clipboard ready to be pasted.

Function 9–48: select

(tell *text–editor* mid:select *marker1 marker2*)

Returns*: t and selects contents between *marker1* and *marker2

Once the material is selected, you can use the text–editor to copy, cut or paste it.

Why would you use the text–editor instead of just setting markers and directly getting the substring, deleting it, and inserting it? For one thing, you might want to show the material being selected on screen. But, more important, the text editor handles all of the "details," such as updating the font bar, etc.

Function 9–49: deselect

(tell *text–editor* mid:deselect)

Returns: t. Deselects any selected material

This function returns t no matter what.

Function 9–50: split
(tell *text–editor* mid:split)
Splits the component into two components at the text caret
Function 9–51: hyphen
(tell *text–editor* mid:hyphen *&keyword* :show :set :clear)
Returns: t

The keyword *:show* works only if there isn't any text selected. It puts a message in the message bar of the document window indicating the hyphenation points of the word in which the text caret sits. The keyword *:set* forces a hyphenation point (but not necessarily a hyphen, of course) where the text caret is. The keyword *:clear* clears the word of all hyphenation points.

Since these functions all return t, suppose you want to access the hyphenation points programatically, rather than displaying them to the user? Remember that hyphenated characters are their own special token which you can get through the *marker get–substring* method.

9–31 get–hyphenation–points

```
(defun get–hyphenation–points (m1 m2)
; Returns list of hyph–chars between two markers
  (tell m1 mid:get–substring m2 '(:hyph–char)))
```

Misspelled words are flagged internally by having their last character set as a hyphenated character. So, the above function will return the last characters of any misspelled words. You could fix this by altering the function so it takes one word at a time, and then checking the hyphenated characters to make sure they're not the last character in the word. You could also take advantage of this fact to write a function that checks to see if a word has been flagged as misspelled:

9–32 misspelled–word

```
(defun misspelled–word (m)
; m is a marker at the end of a word
  (let (m1)
    (setq m1 (tell m mid:copy))
    ; move m1 back one token
    (tell m1 mid:move–by –1)
    (tell m1 mid:get–substring m '(:hyph–char))
))
```

You could now write a routine that looks through a component or a document, moving the marker by word-endings, and thus compiling a list of misspelled words (or a list of markers pointing at misspelled words ... or both.)

Similarly, you could rebind the word-ending characters (space, punctuation, hard return, etc.) so that before inserting themselves, they check the character before them to see if it's

hyphenated (i.e., misspelled). If it is, then perhaps the system beeps at you, or adds the word to a word list, etc.

Function 9–52: spell

(tell *text–editor* mid:spell)

***Returns*: t and runs the spell checker**

This does exactly what a user does when she runs the spell checker by making a popup menu choice.

Function 9–53: shape

(tell *text–editor* mid:shape)

***Returns*: t and runs the text shaping function**

This puts the user into text-shaping mode, exactly as if she invoked it through the popups. The programmer, however, can shape text directly by using the component editor (discussed below).

Function 9–54: create–ref

(tell *text–editor* mid:create–ref :page "page")

Inserts page reference

This inserts a reference to the current page, the same as manually creating one.

Function 9–55: inline

(tell *text–editor* mid:inline *name*)

Creates inline, or converts selected text to inline *name*

E.g. *(tell *text–editor* mid:inline "para")*
Creates inline named "para"

If text is selected, the above function converts it to an inline; if none is selected, it creates an empty inline.

Function 9–56: convert–case

(tell *text–editor* mid:convert–case &*keyword*)

Converts case of selected text to *keyword*

E.g. *(tell *text–editor* mid:convert–case :lower nil)*

The choices of argument are *:upper*, *:lower*, and *:initial*. The last argument determines whether accents should be stripped; if it's non-nil, the accents will be stripped.

Function 9–57: search

(tell *text–editor* mid:search)

Opens search prop sheet

Function 9–58: replace

(tell *text–editor* mid:replace)

Opens replace prop sheet

If any text is selected, it will be entered into the search and replace prop sheet's Search String field.

Function 9–59: auto–ref

(tell *text–editor* mid:auto–ref)

Creates auto reference

Function 9–60: goto

(tell *text–editor* mid:goto &*keyword*)

Takes you to the next *argument* object

E.g. *(tell *text–editor* mid:goto :frame)*
Takes you to next frame

The argument options are *:frame*, *:autonumber*, *:index*, *:reference*, or *:inline*.

Function 9–61: create

(tell *text–editor* mid:create &*keyword name*)

Creates the named thing at the text caret

E.g. *(tell *text–editor* mid:create :frame "Rule")*
Creates a frame named "Rule"

This powerful function allows you to insert, at the text caret, a frame, index token, autonumber, or inline. If you give the name of an object that already has a master, then an instance of that object is inserted; if there is no master, then a default object is created with the name you've supplied. Here is a list of what you can create:

Keyword	Argument	Comment
:frame	name	If frame has shared or initial contents, those are created along with it
:inline	name	
:index	string	String is used as entry in index
:autonumber	name, level	Level is zero-based, so *(tell *text-editor* mid:create :autonumber "chaptnumb" 1)* creates second-level autonumber

Working with components

As with the text editor, you can work with components either directly or by invoking the component editor. The component editor allows the programmer to easily mimic the interactions of an end user with the system; working directly with components provides a finer degree of control but also requires the programmer to pay more attention to getting all the details right.

This time we'll begin by discussing how to work with components directly since there are things we're going to want to do right away — such as adjust component properties — that are easy to do using the direct method. Then we'll discuss the component editor.

Working directly with components

You work directly with components first by getting a component object and then by telling it do something.

The first task is to get a component object.

Function 9–62: doc–point–cmpn

(doc–point–cmpn)

***Returns*: component the text caret is in**

E.g. *(doc–point–cmpn)*
Returns current component

If the text caret is in an inline, *(doc–point–cmpn)* will give you the inline component.

Function 9–63: doc–point–top–cmpn

(doc–point–top–cmpn)

***Returns*: the top-level component the text caret is in (the one named in the component bar)**

E.g. *(doc–point–top–cmpn)*
Returns current top component

This gives the current top-level component (the one that gets listed in the component bar) even if you are currently in an inline.

Here is a list of the properties that can be retrieved and/or set, arranged in the order in which the end user encounters them on the component property sheet. (*MTIR* refers to whether the properties are available on *m*asters, *t*op-level components, *i*nlines or *r*ows of tables. Rsu's are Interleaf's standard of measurement; there are 1,228,800 rsu's in an inch.)

Format		Returns	Can be set
:name	MTIR	cmpn's name	✓
:doc–font	MTIR	base font for cmpn	✓
:top–margin	MTR	top margin, in rsu's	✓
:bottom–margin	MTR	bottom margin, in rsu's	✓
:left–margin	MT	left margin, in rsu's	✓
:right–margin	MT	right margin, in rsu's	✓
:initial–indent	MT	amount of initial indent, in rsu's	✓
:initial–in-dent–count	MT	number of initial lines to be indented	✓
:line–spacing	MT	line spacing, in rsu's	✓

Format		Returns	Can be set
:line–spacing–lines	MT	if t, line spacing is specified in lines	✓
:line–spacing–fixed	MT	t if line spacing is based on cmpn font size	✓
:alignment	MT	alignment— :left, :right, :center, :inner, :outer, :justified	✓
Page			
:begin–new–page	MT	forces cmpn to begin new page	✓
:begin–new–column	MT	forces cmpn to begin new column	✓
:allow–break–before	MTR	t if break is allowed before cmpn	✓
:allow–break–within	MTIR	t if break is allowed within cmpn	✓
:allow–break–after	MTR	t if break is allowed after cmpn	✓
:orphan–count	MT	avoid breaks that leave this number (1–16) of lines at the beginning of cmpn	✓
:widow–count	MT	avoid breaks that leave this number (1–16) of lines at the beginning of cmpn	✓
:straddle–columns	MT	t if cmpn straddles columns	✓
Custom			
:read–only	MTIR	forces cmpn to be read only	✓
:toc–document–name	MTI	name of document to hold the table of content listing	✓
:hyphenation–amount	MTI	how easily to allow hyphenation, from 0 (off) to 10	✓
:track–kern–spaces	MT	kerns spaces, if track kerning is on	✓
:prefix–content	MTI	t if cmpn is shared prefix	✓
:shared–content	MTI	t if cmpn is shared contents	✓

Format		Returns	Can be set
Tab			
:tab–origin–column	MT	t if tabs are relative to the column; nil if relative to margin	✓
:tab–stops	MT	list of cons (location . type) where location is in rsu's and type is :left, :center, :right, :numeric	✓
Profile			
:profile	MT	list of (line–count indent is–right) where line–count is the starting line number (starting with 1), indent is the amount to indent in rsu's, and is–right is t if the indentation is from the right margin	✓
Attr			
:attributes	MTIR	attributes and values for the cmpn	✓
Inline-specific			
:font–inherit	MI	sets font inherit of inlines	✓
Table–specific			
:table–row	R	t if cmpn is a table row	
:row–border–rulings	R	t if show rulings at end of row	✓
:table–header	R	t if use this row as header	✓
:table–footer	R	t if use this row as table footer	✓
:broken–table–row	R	t if row is result of the previous being broken across pages or columns	
:vertical–straddle	R	t if a vertical straddle enters this row from above	
Other			
:force–hidden	MTIR	forces cmpn not to show	✓

As usual, you get these properties with the following function:

Function 9–64: get–props

(tell *cmpn* mid:get–props *&keyword* ...)

***Returns*: requested properties**

E.g. *(tell (doc–point–cmpn) mid:get–props :alignment)*
Gets current component's alignment. Returns, e.g., :left

If you do not specify any keyword, you will get a long list of all the properties.

You can set the properties with a closely related function:

Function 9–65: set–props
(tell *cmpn* mid:set–props &*keyword* ...)
***Returns*: requested properties and sets *cmpn*'s properties**
E.g. *(tell (doc–point–cmpn) mid:set–props :alignment :left)*
Returns :left and makes the current cmpn left-aligned

Some of the properties require additional notes.

Name

You can get a component's name by asking for its *:name* property *(tell (doc–point–cmpn) mid:get–props :name)* or you can use a special function:

Function 9–66: get-name
(tell *cmpn* mid:get–name)
Returns*: the name of *cmpn
E.g. *(tell (doc–point–cmpn) mid:get–name)*
Returns name of current component

Components don't have a set–name method; if you want to set a component's name, you need to use the set–props function, e.g., *(tell (doc–point–cmpn) mid:set–props :name "bullet")*. If you set a component's name to that of an already-existing master, however, it will not change anything else about that component. For example,

```
(tell (doc–point–cmpn) mid:set–props :name "Title")
```
Changes the current component's name to "Title" but leaves its other properties unaltered

If you want to change both the name and the properties, the easiest way to do it is by using the component editor, e.g., *(tell *cmpn–editor* mid:change "subsect3")*, discussed below. (This is an example of the advantages and disadvantages of doing things directly or through the component editor.)

Tab–stops

Asking for the tab stops gives you a list such as the following:

```
; get the tabs
(tell test–cmpn mid:get–props :tab–stops)
```
Returns ((0 . :left) (1228800 . :left) (2457600 . :right) (3686400 . :left))
```
; set the tabs
(tell test–cmpn mid:set–props :tab–stops (list (cons 1000000 :right)(cons
2000000 :center)))
```
Returns ((1000000 . :right) (2000000 . :center))

Remember, if you set the tab stops of a component, you wipe out all the existing tabs. For example, suppose we now get the tab stops of the component named "test–cmpn" used in the example above:

```
; get the tabs
(tell test-cmpn mid:get-props :tab-stops)
Returns ((1000000 . :right) (2000000 . :center))
```

The original tabs have been completely replaced by the new list; all the other tabs have been lost. So, to add a tab without altering any of the original ones, you need a function such as the following:

9–33 add-a-tab

```
(defun add-a-tab (cmpn tab-position tab-type)
  (let (tabs)
    ; get current tabs
    (setq tabs (tell cmpn mid:get-props :tab-stops))
    ; add new tab to list
    (push (cons tab-position tab-type) tabs)
    ; set new tabs
    (tell cmpn mid:set-props :tab-stops tabs)
))
```

To use this function, you would invoke it as follows: *(add-a-tab (doc-point-cmpn) 2000000 :right)*.

Fonts

You have seen that you can change the fonts of any selected text by using the text editor. You could also change fonts more directly by inserting a font token at any marker. Either technique affects the text stream without affecting the properties of the component. To change the component properties, the end user goes to the component property sheets. Programmers can do so more directly.

In Interleaf 5, a font is an object. You can ask a component to hand over its font object through the *get-props* method. Then you can examine those properties, or make a new font object. For example,

```
; get the font object and set df to it
(setq df (tell (doc-point-cmpn) mid:get-props :doc-font))
Returns #<doc-font 0x8c4060>
; look at the font object's properties
(tell df mid:get-props)
Returns (:family "Thames" :bold nil :italic nil :size 12.0 :underline nil
:strikethrough nil :dbl-underline nil :overbar nil :superscript nil :subscript
nil :revision-bar nil :caps nil :pair-kerning t :track-kerning nil :fn-font
#<font-obj 0x98c000> :dictionary #<sdict-obj 0x6d0800> :zoom nil)
; look at a particular property of the font object
(tell df mid:get-props :family)
Returns "Thames"
```

```
; create a new font based on df, with changes (Swiss, 17.5pts, bold,
underlined)
(setq new-font (tell-class doc-font-class mid:new :doc-font df :family
"Swiss" :size 17.5 :bold t :underline t))
Returns #<doc-font 0x8c4180>
:apply new font to component
(tell (doc-point-cmpn) mid:set-props :doc-font new-font)
Returns #<doc-font 0x8c4180>
```

Here are two functions that together use font changes as a sort of interface to other functionality. This case supposes you have a component with a bunch of inlines, and you want the user to be able to move from one inline to another by hitting the tab key. You will signify which inline the user is in by putting it into boldface. If the user is in the last inline, hitting the tab key takes him or her to the first inline. The following way of doing this assumes that you have created a component named "movingbar" that consists of inlines named "choice" and that the first *choice* inline starts off already bolded.

9–34 get-next-choice

```
(defun get-next-choice ()
; moves caret to next inline named choice, puts it into boldface,
; and puts all others into roman
  (let (df bold-font unbold-font next m (c (doc-point-cmpn)))
    ; get current font
    (setq df (tell (doc-point-cmpn) mid:get-props :doc-font))
    ; unbold current cmpn
    ; make bold font
    (setq bold-font (tell-class doc-font-class mid:new :doc-font df
    :bold t))
    ; make unbold font
    (setq unbold-font (tell-class doc-font-class mid:new :doc-font df
    :bold nil))
    ; make current cmpn unbold
    (tell c mid:set-props :doc-font unbold-font)
    ; go to next inline
    (setq next (tell c mid:get-next))
    ; if at end, next is nil
    (if (not next)
      (progn
        ; get first inline
        (setq next (tell (tell c mid:get-top-cmpn) mid:get-child))))
    ; set next inline to bold
    (tell next mid:set-props :doc-font bold-font)
    ; get marker in next inline
```

```
    (setq m (tell next mid:get-marker))
    ; go to that marker
    (doc-goto-marker m)
))
```

9–35 alter–tab

```
(defun new-tab ()
; rebind tab key so it looks for cmpn named movingbar
  ; are we in a component named "movingbar"?
  (if (string= "movingbar" (tell (tell (doc-point-cmpn)
  mid:get-top-cmpn) mid:get-name))
    (get-next-choice)
    ; else tab key should just insert a tab
    (key-insert-tab))
)

; Try it: rebind tab key
(kbd-bind kbd-doc-map "\t" 'new-tab)
```

The following set of functions allow you to toggle between something like a "draft" mode and a WYSIWYG mode. In draft mode, all fonts are replaced by a single font of your choice, in single column.

```
; choose your draft font and size
(setq  *draft-face* "Courier"
       *draft-size* 14)
```

9–36 draft–scan

```
(defun draft-scan (o )
; every cmpn gets sent here for draft formatting
(let (df new-df)
  ; get current doc font
  (setq df (tell o mid:get-props :doc-font))
  ; if there is a doc font, build a new one
  (if df
    (progn
      ; create a new one with different family and size
      (setq new-df
        (tell-class doc-font-class mid:new :doc-font df
          :family *draft-face*
          :size *draft-size*))
      ; set cmpn props to new doc font
      (tell o mid:set-props :doc-font new-df)
      ; save original font as temporary lisp data
```

```
      (tell o mid:put-data :font df)))
))
```

9-37 draft-mode

```
(defun draft-mode ()
; puts document into draft mode, if not already there
  (if (not (tell *document* mid:get-data :draft))
    (progn
      ; turn doc into draft version
      ; get number of cols and put it as temporary data
      (tell *document* mid:put-data :cols
      (tell *document* mid:get-props :columns))
      ; set doc to one col
      (tell *document* mid:set-props :columns 1)
      ; scan doc to set fonts to draft fonts
      (doc-scan-class-apply 'draft-scan (list doc-cmpn-class))
      ; update the doc
      (doc-flush-queue)
      ; put some data so we can check whether it's in draft mode
      (tell *document* mid:put-data :draft t)
    ))
)
```

9-38 undraft-mode

```
(defun undraft-mode ()
  ; if there's saved data, read number of columns and restore
  (if (tell *document* mid:get-data :cols)
    (tell *document* mid:set-props :columns
      (tell *document* mid:get-data :cols)))

   ; scan the doc and restore any saved fonts
  (doc-scan-class-apply '(lambda (o)
    (if (tell o mid:get-data :font)
      (tell o mid:set-props :doc-font (tell o mid:get-data :font))))
    (list doc-cmpn-class))
  ; update the doc
  (doc-flush-queue)
  ; tell it it's not in draft mode any more
  (tell *document* mid:put-data :draft nil)
)

; bind some keys -- ^Xr to draft it, ^XR to undraft it
(kbd-bind kbd-doc-map "\^Xr" 'draft-mode)
(kbd-bind kbd-doc-map "\^XR" 'undraft-mode)
```

Attributes

Interleaf 5 allows the user — or the programmer — to attach any information (in the form of *attributes*) he or she wants to any Interleaf object. The main purpose of this for the end user is to allow the end user to automatically assemble different versions of a document based upon the information attached to the objects. This is known as *effectivity* or *conditional document assembly.*

For the programmer, however, this capability is even more useful, for it provides a very convenient way to attach data that a *program* might need. You can even use the object's attributes to store code that then gets *eval*'ed. (You can also use Lisp data, as explained in a previous chapter.)

Function 9–67: get-attrs

(tell *cmpn* mid:get–attrs)

***Returns*: list of attributes and values**

E.g. *(tell (doc–point–cmpn) mid:get–attrs)*
Returns "(("prod#""123.4a" "5.5")("security""top"))

What *get–attrs* returns is a list of lists. Each inner list consists of the attribute's name and all of its values. If an attribute has no value, neither the attribute nor its value is returned.

You can get a specific attribute by supplying its name as an argument:

```
(tell (doc–point–cmpn) mid:get–attrs "prod#")
Returns, e.g., ("123.4a""5.5" ), or nil if no value
```

In this example, the attribute *prod#* has two values. It is important to note that if an object has no value for a particular attribute, that attribute does not get reported at all. For example, if the next component's "do" attribute were blank, the attribute list might look like this: *(("nextcmpn" "code") ("prod#" "123.41" "5.5")).* And if there are no values for any attribute for an object, when you ask to see its attributes, you will get a nil returned to you.

To find out what all the attributes are that have been defined in a document, you have to ask the document icon:

```
(tell (doc–current–icon) mid:get–props :attributes–control)
Returns (("prod#" 2) ("security" 1))
```

In this case, the document has *prod#* defined as an attribute that can have two values, whereas *security* can only take one value.

If you want to add a new attribute to a document (not fill in an attribute value), you only have to set an object to include that attribute; it will automatically show up on the document property sheet.

set–attributes

```
(defun set–attributes (cmpn attr–list)
  (tell cmpn mid:set–props :attributes attr–list)
```

```
)
; try it out
(set-attributes (doc-point-cmpn)
  (list (list "date" "11-23-93") (list "products" "Zizzy Scoop" "The
Abdomifier")))
```

To use this function, you need to construct a list of the attributes you want to set. The list contains a list for each attribute you want to set (consisting of the attribute name and all the values you want to set.) The list has to be complete. For example, if a component has attributes set for "date" and "place," and you give *set-attributes* a list that contains a value only for "date," the "place" attribute values will be wiped out.

To change just one attribute value, you can use the following:

9-39 alter-attr-value

```
(defun alter-attr-value (cmpn attr)
; set a new attr value without disturbing other current ones
  (let (current-attrs )
    ; get current attribute list
    (setq current-attrs (tell cmpn mid:get-props :attributes))
    ; is new one is already on list?
    (if  current-attrs
      (progn
        (setq current-attrs (delete attr current-attrs :test
          '(lambda (a b) (string= (car a) (car b)))))))
        ; add new value to list
        (push attr current-attrs)
        ; set cmpn to new attr list
        (tell cmpn mid:set-props :attributes current-attrs)
))

; try it
(alter-attr-value (doc-point-cmpn) (list "product#" "second"))
```

You can, of course, set control expressions through Interleaf Lisp that will look at every object's attributes and see if the object ought to be shown or be hidden; this simulates the way end users use effectivity.

9-40 set-control-expression

```
(defun set-control-expression (expr)
; applies expr string to document
  (let ((doc (doc-current-icon)))
    ; set the control expression
    (tell (doc-current-icon) mid:set-props :local-control-expression
```

```
    expr)
    ; tell it to pay attention to its control expression
    (tell (doc-current-icon) mid:set-props
    :local-control-expression-enabled t)
))
; try it out
(set-control-expression "$name # order-number")
```

This allows you to create an easier interface for altering attributes. The following creates an input stickup with the current attribute value already in the input area. If you enter new data into the stickup, it gets applied to the component.

9–41 attribute–ui

```
(defun attribute-ui (c attr)
; shows value of attribute attr of cmpn c
  (let (new-value value)
    ; get value of attribute
    (setq value (tell c mid:get-attrs attr))
    ; if has no value, call it nil
    (if (not value) (setq value "<Nil>"))
    ; prompt for new value and show old one in stickup field
    (setq new-val
      (stk-open-prompt (format nil "Enter new value of ~A" attr)
    :initial-prompt value))
    ; set attr to new value using previously created function
    (if new-val
      (alter-attr-value c (list attr new-val)))
))

; try it
(attribute-ui (doc-point-cmpn) "product#")
```

With attributes, you can take the moving bar menu we began to create and make it come alive. Using Interleaf 5's end-user interface, create a component called "movingbar" that consists of inlines called "choice." Create an attribute called "do." For each of the inlines, give a value to the *do* attribute that is in fact a piece of Interleaf Lisp code. For example, one might be *(stk-open "Howdy")* and another might be *(tell *text-editor* mid:save)*. Now we're going to rebind the return key so that it will check to see if we're in a *choice* inline, and, if so, it will try to execute whatever line of code is stored in the *do* attribute.

Function 9–68: doc–eval–attribute

(doc-eval-attribute *cmpn attr*)

***Returns*: nil if no attribute**

E.g. *(doc-eval-attribute (doc-point-cmpn) "do")*
Evals the "do" attribute of current component and returns value

This function gets the value of the attribute you want and then assumes that that value actually is some Lisp code which it then *eval*s. So, if you had a component with an attribute called "code" which consists of the value *(tell *cmpn-editor* mid:create "signature")*, *doc-eval-attribute* would *eval* the value and create a new component named *signature*.

You could now use some of the functions developed in this chapter to create a moving-bar menu which allows you to tab through various fields and use the return key to execute a choice. First, *eval* the moving-bar code above which rebinds the tab key so that it cycles through inlines in a component. Then, *eval* the following code which rebinds the return key so that if the current component is named *choice* and has an attribute named "do," it will try to *eval* the value of "do"; if it has no value, it inserts a hard return.

9-42 moving-bar-action

```
(defun moving-bar-action ()
  (let (to-do (c (doc-point-cmpn)))
    ; get contents of "do"
    (if (not (doc-eval-attribute c "do"))
      (key-insert-return))
))
```

9-43 new-return

```
(defun new-return()
; rebind return key so it looks for inline named "choice"
  (if (string= "choice" (tell (doc-point-cmpn) mid:get-name))
    (moving-bar-action)
    ; else
    (key-insert-return))
)

; rebind return key
(kbd-bind kbd-doc-map "\r" 'new-return)
```

The programmer can still get at any object hidden through effectivity. Navigating *:along :structure* will bring you to every hidden element. You can look at each object to see if it is visible. For example,

9-44 count-hidden-elements

```
(defun count-hidden-elements ()
  (let ((hidden-ctr 0))
    (doc-scan '(lambda (obj level)
      (if (not (tell obj mid:is-effective))
        (inc hidden-ctr))))
  hidden-ctr
))
```

```
; try it
(stk-open (itoa (count-hidden-elements)))
```

Force-hidden

While effectivity provides a very powerful mechanism for storing Interleaf Lisp data, it is often not very useful to the programmer as a way of showing or hiding document objects, precisely because it *is* a useful way for end-users to show and hide content. Because the programmer cannot predict what control expression the end user may have in place, the programmer would have to preserve that control expression when adding or deleting new conditions. That would mean lots of careful string handling.

Fortunately, Interleaf Lisp provides another, more direct, technique. Any document object can be *force-hidden* which means that it will not be visible. Because it works with text and with graphics, force-hidden is useful not only for changing the appearance of documents but also when building document-based interfaces.

Here is a function that hides or shows all the frames in a document:

9-45 hide-or-show-frames

```
(defun hide-or-show-frames (toggle)
; if toggle is t, hides frames; if nil, shows them
  (doc-scan '(lambda (obj level)
      (if (is-of-class obj doc-frame-class)
        (tell obj mid:set-props :force-hidden toggle))))
)
;try it
(hide-or-show-frames nil)
```

As you can see, force-hidden provides a powerful capability.

In fact, since the frames have been hidden through force-hidden, there is no user interface that will bring them back. The following routine will unhide anything hidden by force-hidden. (It leaves hidden anything hidden because of the effect of a control expression.)

9-46 force-unhide-all

```
(defun force-unhide-all ()
  (doc-scan '(lambda (obj level)
    (tell obj mid:set-props :force-hidden nil)))
)
```

You can use force-hidden to provide an automated outliner. In the following example, ^X- causes every component of less importance than the current one to hide itself; ^X+ shows all the lesser components you've hidden. "Lesser" means it is of lesser importance in the structure of the document, To determine this, the function uses a simple and easily fooled set of "heuristics." For example, if component A is in a larger type size than compo-

nent B, it assumes that A is greater than B in terms of its relative importance in the document. Likewise, being centered and bolded makes A greater than B if B either isn't centered or isn't bolded. Clearly you could come up with your own "heuristics" that would make the following freeform outliner work better. (Be prepared to use example 9–46 — *force–unhide–all*— to restore your document as you experiment with this code!) (Note: The following leaves table rows untouched because if you try to get their doc font, they give you nil; you instead have to look inside the individual cells.)

9–47 is–lesser–than

```
(defun is-lesser-than (c size center bold indent)
; "heuristics" to decide if c is lesser than where we started
  (let (this-size (lesser nil) this-bold df)
  ;get new fontsize
  (setq df (tell c mid:get-props :doc-font))
  (setq this-size (tell df mid:get-props :size))
  (setq this-bold (tell df mid:get-props :bold))
  ; if this one is smaller type size
  (if (< this-size size)
    (setq lesser t))
  ; if original is centered and bold and this one isn't
  (if (and (equal :center center) bold
    (or (not (equal :center (tell c mid:get-props :alignment)))
      (not this-bold)))
        (setq lesser t))
  ; if this one is indented and equal or smaller
  (if (and (> (tell c mid:get-props :left-margin) indent)
    (<= this-size size))
      (setq lesser t))
  lesser
))
```

9–48 outline–hide–levels

```
(defun outline-hide-levels ()
; hides any cmpn less than current cmpn
  (let (font-size bold indent centered c doc-font)
  ; get current cmpn's parameters
  (setq c (doc-point-cmpn))
  ; get doc font so can get size and boldness
  (setq doc-font (tell *text-editor* mid:get-props :doc-font))
  (setq font-size (tell doc-font mid:get-props :size))
  (setq bold (tell doc-font mid:get-props :bold))
  (setq indent (tell c mid:get-props :left-margin))
  (setq centered (tell c mid:get-props :alignment))
```

```
  ; loop hiding anything "less" than current cmpn
  ; get first cmpn
  (setq c (tell-class doc-cmpn-class mid:get-first))
  (while c
      ; skip table rows and check for lesserness
    (if (and (not (tell c mid:get-props :table-row))
      (is-lesser-than c font-size centered bold indent))
      (progn
        (tell c mid:set-props :force-hidden t)))
        (setq c (tell c mid:get-next))
      )
  (doc-flush-queue)
))
```

9–49 outline–show–levels

```
(defun outline-show-levels ()
; shows any cmpn greater than current cmpn
(let (font-size bold indent centered c pt doc-font)
  ; get current position
  (setq pt (doc-point-marker))
  ; get current cmpn's parameters
  (setq c (doc-point-cmpn))
  ; get doc font so can get size and boldness
  (setq doc-font (tell *text-editor* mid:get-props :doc-font))
  (setq font-size (tell doc-font mid:get-props :size))
  (setq bold (tell doc-font mid:get-props :bold))
  (setq indent (tell c mid:get-props :left-margin))
  (setq centered (tell c mid:get-props :alignment))
  ; loop hiding anything "less" than current cmpn
  ; get first cmpn
  (setq c (tell-class doc-cmpn-class mid:get-first))
    (while c (inc ctr)
    ; skip table rows and check for lesserness
      (if (and (not (tell c mid:get-props :table-row))
        (is-lesser-than c font-size centered bold indent))
        (progn
          (tell c mid:set-props :force-hidden nil)))
    (setq c (tell c mid:get-next :along :structure))
  )
  (doc-flush-queue)
))
```

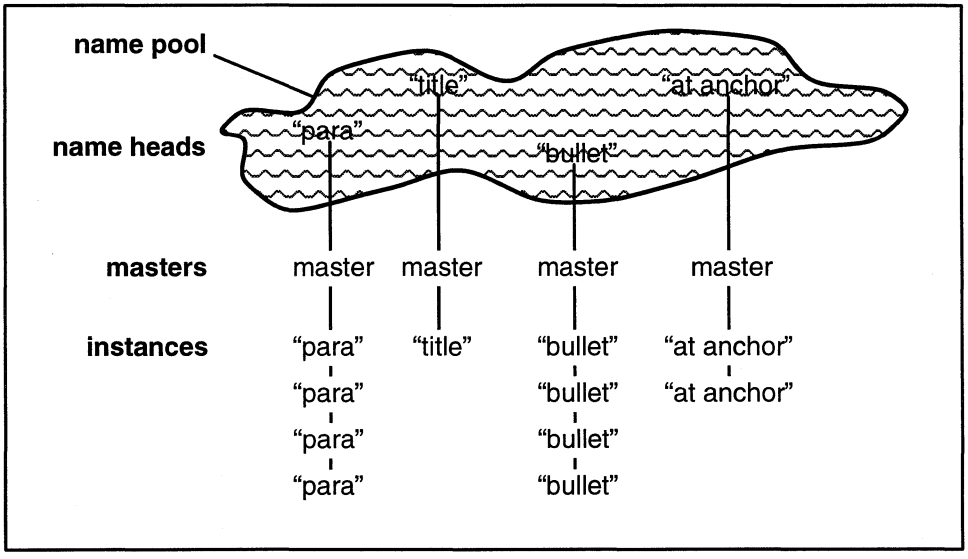
```
; bind it to some keys
(kbd-bind kbd-doc-map "\^X+" 'outline-show-levels)
(kbd-bind kbd-doc-map "\^X-" 'outline-hide-levels)
```

Remember, while effectivity is one way of attaching data to objects, *put-saved-data* is another.

Masters

You've learned how to alter the properties and content of individual objects. But Interleaf 5 provides masters, not just instances. A master is a component's definition. When the user creates a new instance of a component by using the Create menu in the component bar, the system creates an instance that exactly reflects the master. (If the user creates a new instance by hitting the ^J key, the new instance is modelled on the current component, not on the master; the user may have adjusted the current component without updating the master.) With Interleaf Lisp, you can manipulate a master just the way you manipulate instances.

First you have to get the master. To do that, you have to check the *name pool* of the class of object — in this case, component — you're looking for. The name-pool consists of the names of the objects in the document. The name pool has a *name-head* which is the head of the chain. You can use the name pool to get the master. You can also use it to visit every object with that name.



Function 9–69: name–find–pool

(name–find–pool *class &optional parent*)

Returns*: name pool of *class* in *parent

E.g. *(name–find–pool doc–cmpn–class *document*)*
Returns the name pool for all components in the current document

The name pool for components contains all the components in the document. To traverse this list, however, you have to find its head:

Function 9–70: name–find–head

(name–find–head *class name &optional parent*)

Returns*: the head of the name pool for *class* by the name of *name* in *parent

E.g. *(name–find–head doc–cmpn–class "para" *document*)*
Returns head of the name pool for all components named "para" in the current document

The object classes for which you can find the name pool and name head include the following:

- doc–cmpn–class
- doc–frame–class
- doc–table–class
- doc–table–cell–class
- doc–autonum–stream–class
- doc–pagenum–stream–class
- dg–named–class
- rev–edit–class
- rev–version–class

Now that you have the name header, you want to do something with it. You can get the master if you can get the head by asking for the head's child *:along :structure*:

9–50 get–master

```
(defun get-master (class name)
; get the master of object
  (let (head)
    (setq head (name-find-head class name))
    (if head
      (tell head mid:get-child :along :structure))
))
```

For example,

(get-master doc-cmpn-class "bullet")
Returns the master for the component "bullet"

Now that you have the master, you can alter it by getting and setting its props just as with any component. Of course, altering the master won't alter any of the existing instances, although the next instance created will reflect the changes. (You'll see how to change the instances in a minute.)

If you want to confirm that the object in your hands is a master, you can use the following function:

Function 9-71: is-master
(tell *object* mid:is-master)
***Returns*: t if object is a master; otherwise, nil**
E.g. *(tell (doc-point-cmpn) mid:is-master)*
Always returns nil because the current component is never the master

Suppose you want to get all the masters of a certain class in a document.

9-51 get-masters

```
(defun get-masters (class)
  (let (pool name-head  (master-list nil) child)
    (setq pool (name-find-pool class))
    ; get first
    (setq name-head (tell pool mid:get-child))
    (while name-head
      ; get master from name head
      (setq child (tell name-head mid:get-child :along :structure))
      (if child
        (push child master-list))
      (setq name-head (tell name-head mid:get-next))
    )
  master-list
))
```

If you only want to see the masters that actually have instances, you could process the list of masters you just created:

9-52 get-used-masters

```
(defun get-used-masters (class is-used)
; returns list of masters, optionally only those with instances
  (let (name-head  (master-list nil) child (used t))
    (setq pool (name-find-pool class))
    ; get first
    (setq name-head (tell pool mid:get-child))
```

```
    (while name-head
      (setq child (tell name-head mid:get-child :along :structure))
      ; if we want only used masters, find out if it's used
      (if is-used (setq used (tell name-head mid:is-used)))
      (if (and child used)
        (push child master-list))
      (setq name-head (tell name-head mid:get-next))
    )
  master-list
))
```

This slight variation on *get-masters* has an additional argument. If it is *t*, then the function will only return masters that have at least one instance.

You can use the name pool to visit every instance of a master as well. This makes it a fast way to apply changes to every component with a certain name, for example.

9–53 global-apply

```
(defun global-apply (name fnc &optional value)
  (let (head m)
    (setq head (name-find-head doc-cmpn-class name *document*))
    (setq m (tell head mid:get-child :along :name))
    (while m
      (tell m mid:set-props fnc value)
      (setq m (tell m mid:get-next :along :name)))
))
```

For example, *(global-apply "para" :alignment :right)* will right align every instance of "para" and will set its master to right aligned as well.

You could also use these functions to visit every component in your document and change the fonts from, say, Thames to Swiss. But be warned — if you change your mind afterwards, you'll have to go back through and restore the original fonts by hand. (You could expand this function so that it stores the original fonts as an attribute of each component — or, better, as lisp data via *mid:put-data* — so you could undo it programmatically later on.)

9–54 change-all-fonts

```
(defun change-all-fonts (family size)
  (let (pool child c (name-list nil) df newfont)
    ; post a warning
    (if (not
      (stk-open
        (format nil "Are you sure you want to change all fonts\nin ~D
to\n~Dpt. ~A? NOT recoverable!"
        (tell (doc-current-icon) mid:get-name) size family) :yes-no))
```

```
    (quit))
  (setq pool (name-find-pool doc-cmpn-class *document*))
  ; get list of name heads
  (setq child (tell pool mid:get-child :along :name))
  (while child
    (push child name-list)
    (setq child (tell child mid:get-next))
  )
  ; go through list of name heads and find all instances
  (while (setq m (pop name-list))
    ; go through all instances
    (setq c (tell m mid:get-child))
    (while c
      ; create appropriate doc font
      (setq df (tell c mid:get-props :doc-font))
      (setq newfont (tell-class doc-font-class mid:new :doc-font df
      :family family :size size))
      ; apply new font
      (tell c mid:set-props :doc-font newfont)
      (setq c (tell c mid:get-next))))

))

;try it
(change-all-fonts "Swiss" 14.5)
```

Component Editor

One of the objects available to the programmer within a document is the *component editor*, much like the text editor and other editors. This provides a convenient way to accomplish many component-oriented tasks.

You can tell the component editor to *get-props* and *set-props*:

:caret-direction	If *:next*, component caret will appear just after the component with the text caret; if *:previous*, it will appear just before it
:cmpn-bar-width	Width of the component bar in screen pixels
:view-cmpn-bar	If t, the component bar shows. If nil, it doesn't show. If *string*, the component bar will show the value of the attributer that matches the string

You can mimic the end user's interaction with the component bar menus by using the component editor, as well as do more.

Function 9–72: get–child

(tell *cmpn–editor* mid:get–child)

***Returns*: first top–level component in the document**

This does the same as *(tell *document* mid:get–child)*

The component editor cares whether anything is selected in the component bar. In this it mirrors programmatically the end user's experience with the software. Fortunately, there are a number of ways of controlling what is selected.

Function 9–73: select

(tell *cmpn–editor* mid:select *&optional arguments*)

[no arguments]	Selects component the mouse cursor is pointing at — in the text or in the component bar
component	Selects the specified component
name	Selects all components with that name
:all	Selects all components (or all in current microdocument)
:table *name*	Selects table named *name*
:begin	Selects all from first through current component
:end	Selects all from current component through end
:toggle	Selects what was deselected (except hidden components) and deselects what was selected.
:page *page–number*	Selects all components from current component to the specified page. Page numbering is zero-based; the first page of the document is page 0, even if in a book. If no number is supplied, user will be prompted for one

Unfortunately, there is no built-in way to get a list of all the selected components. The following function should suffice, however:

9–55 get–selected–cmpns

```
(defun get–selected–cmpns ()
  (let (c (sel–list nil))
    ; go to first cmpn
    (setq c (tell *cmpn–editor* mid:get–child))
    ; loop through document
    (while c
      (if (tell c mid:is–selected)
        (push c sel–list))
      (setq c (tell c mid:get–next)))
  sel–list
))
```

Function 9–74: deselect
(tell *cmpn–editor* mid:deselect *&optional argument*)
Deselects specified components
E.g. *(tell *cmpn–editor* mid:deselect :all)*
Deselects all components

Without any arguments, *(tell *cmpn–editor* mid:deselect)* will deselect whichever component the mouse cursor is over. With *:all*, all components are deselected. Or you can specify which precise component you would like deselected, i.e., *(tell *cmpn–editor* mid:deselect cmpn)*.

Function 9–75: cut
(tell *cmpn–editor* mid:cut)
Cuts selected components

If there are no components selected, you'll get an error message. Otherwise, all the selected components will be cut and written to the clipboard, which means they can be pasted using the *paste* method.

Function 9–76: copy
(tell *cmpn–editor* mid:copy)
Copies selected components

Copies any selected components onto the clipboard from which they can be pasted.

Function 9–77: paste
(tell *cmpn–editor* mid:paste)
Pastes whatever was selected on the clipboard

If anything is selected in the document, you'll get an error message. If anything is selected on the clipboard, it will be pasted at the component caret. If nothing is selected on the clipboard, there will be a message to that effect put into the document window's message line.

You can control whether the paste occurs above or below the component with the component caret by setting the props of the component editor. For example:

```
(tell *cmpn–editor* mid:copy)
Copies selected components to clipboard
(tell *cmpn–editor* mid:set–props :caret–direction :next)
Sets caret beneath the component
(tell *cmpn–editor* mid:paste)
Pastes beneath the component
(tell *cmpn–editor* mid:set–props :caret–direction :previous)
Sets caret before the component
(tell *cmpn–editor* mid:paste)
Pastes above the component
```

Notice that only if you have *multi-paste* set to true can you paste twice in a row without re-copying. This variable is part of your profile; look in the *init.lsp* file in your profile drawer, or use your *Profile Tool*.

Function 9–78: create

(tell *cmpn–editor* mid:create *&optional object*)

Creates an instance of the object

E.g. *(tell *cmpn–editor* mid:create "para")*
Creates an instance of "para"

The optional object can be a component's name or a master component. If it is nothing, then an instance of the current component will be created.

This simple function can be used very effectively. Suppose, for example, you create an instance of "list." This may be a component that has a shared prefix which consists of a frame with a graphic of a hand and a microdocument with an autonumber. The prefix may be tagged for a table of contents. With one command, you create an instance that contains all this information, that may cause multiple cross references to be updated across a network, that is properly numbered and listed in the table of contents. That this is possible with a single command is testimony to the power of document object-oriented programming.

Here is a very function that combines several functions. It creates an instance of whatever component is stored in the attribute "nextcmpn" when the user types escape–J. So, for example, you could store "para" as the value of "nextcmpn" for a subhead component, so that when you're in a subhead you can type escape–J and get an instance of "para."

9–56 next–cmpn

```
(defun next–cmpn ()
  (let (attr (c (doc–point–cmpn)))
    ; get attributes
    (setq attr (tell c mid:get–attrs "nextcmpn"))
    ; if no value of "nextcmpn" attr, set name to current cmpn
    (if (not attr)
      (setq attr (tell mid:get–name)))
    ; insert after current cmpn
    (tell *cmpn–editor* mid:set–props :caret–direction :next)
    ; create the new cmpn
    (tell *cmpn–editor* mid:create attr)
))
; bind it to escape–J
(kbd–bind kbd–esc–map "j" 'next–cmpn)
```

With a little more work, you can allow the user to put a series of component names in the *nextcmpn* attribute so that escape–J will create all the listed components.

Function 9–79: change

(tell *cmpn–editor* mid:change *cmpn*)

Changes the selected components into *cmpn*

E.g. *(tell *cmpn–editor* mid:change "para")*
Changes all selected components to "para"

The change function requires that some components be selected.

Function 9–80: join

(tell *cmpn–editor* mid:join)

Joins previous and next components

The previous function requires that nothing be selected.

Function 9–81: goto

(tell *cmpn–editor* mid:goto &keyword master)

***Returns*: Moves caret to object and returns the object**

E.g. *(tell *cmpn–editor* mid:goto :next "para")*
Goes to next instance of component named "para" and returns that component

The keywords allowable here are *:first*, *:last*, *:next* and *:previous*.

Function 9–82: open–props

(tell *cmpn–editor* mid:open–props)

***Returns*: Opens property sheet for selected component and returns the window object**

There must be one and only one component selected for this function to work, just as when it is invoked by an end user.

Pages

Pages are fleeting in Interleaf 5. Everytime you enter a character, it's entirely possible that you will be altering many pages, for your character may cause the document to repaginate itself. Interleaf 5 is built around the *structure* of documents — the relationship of the various components and other objects — not around transient pages. So there is not much emphasis put on pages in Interleaf Lisp.

***Function 9–83:* doc–point–page**

(doc–point–page)

***Returns*: current page**

This gives you the current page. You can then get, but not set, the following properties:

:offset	Page number. The first page of the document is page 0.
:pixel–left–offset	Number of pixels the page is from the left side of the window
:pixel–top–offset	Number of pixels the page is from the top of the window
:visible	If any portion of the page is visible on screen (or very close) — i.e., if it's in the document window -— this is t.

One of the most common uses of pages is to get the page number of a particular object. This is easily done by using the following two functions.

***Function 9–84:* doc–page–of**
(doc–page–of *object*)
***Returns*: page that *object* is on**

This returns the page object, but not yet the page number.

***Function 9–85:* doc–page–number–of**
(doc–page–number of *page*)
Returns*: page number of *page

This takes a page object and tells you what its page number is. The page number is the actual page number; if the object is in a book so that the page number is influenced by other chapters before it, the number returned will be the appropriate one.

These two functions can be combined into one:

9–57 get–page–number–of

```
(defun get-page-number-of (obj)
; returns page obj is on
  (doc-page-number-of (doc-page-of obj))
)
```

You can also navigate through pages. The doc–page–class can be used to get to the first or last pages:

***Function 9–86:* get–first**
(tell–class doc–page–class mid:get–first)
***Returns*: first page object**
***Function 9–87:* get–last**
(tell–class doc–page–class mid:get–last)
***Returns*: last page object**

You can go to a particular page by using the document's *open* method:

9–58 goto–page

```
(defun goto-page (document page-number)
  (tell document mid:open :page page-number)
  )

  ; try it
  (goto-page (doc-current-icon) 357)
```

You can see how this simple function can serve as the basis for a powerful hypertext capability. (If the page number is lower than the number of the first page in the document, this function will take you to the first page; if the number is higher than the last page, it takes you to the last page.)

If you want to go to a particular object, but you don't know, or can't predict, where the object is, you can use the function *get–page–number–of*:

9–59 goto–object–page

```
(defun goto–object–page (document object)
; uses previously–defined function
  (tell document mid:open :page (get–page–number–of object))
  )
```

Notice that with Interleaf 5's power, you can build a hypertext system that works even as the objects you're linking to are moved around.

This function leaves the text caret at the top of the page. If you want to leave it immediately before the object itself, you can use the following function:

9–60 goto–object

```
(defun goto–page (document object)
  ; go to page
  (tell document mid:open :page (get–page–number–of object))
  ; go to object on page
  (doc–goto–marker (tell obj mid:get–marker))
  )
```

If you want to leave the caret immediately after the object, give *get–marker* an argument of t: *(doc–goto–marker (tell obj mid:get–marker t))*.

Columns

Columns are short-lived in Interleaf 5. Whenever the pagination changes, so that text or graphics go from one column to another, the old column object is destroyed and a new one is created.

You can get the following properties of columns:

:type	One of: *:text, :column–text, :straddle–text, :top–frames–only, :follow–anchor–straddle–frames, :follow–text–straddle–frames, :bottom–straddle–frames, :underlay–frames, :overlay–frames, :left–side–frames, :right–side–frames*
:visible	t if any part of the column is visible on screen, although may be true if a column is off to the left or right side of a window
:at–left–margin	t if column is leftmost on page
:at–right–margin	t if column is rightmost on page
:width	Width, in rsu's

:vertically–justified	t if vertically justified
:feathered	t if line spacing will be tinkered with to achieve vertical justification
:height	Height, in rsu's, of the column
:left–offset	Distance, in rsu's, from the left of the page to the left margin of the column
:top–offset	Distance, in rsu's, from the top of the page to the top of the column

You can navigate through columns, but otherwise you can't do much with them in Interleaf 5 (because columns are such transient objects).

***Function 9–88:* get–next**
(tell *column* mid:get–next)
***Returns*: next column**
***Function 9–89:* get–previous**
(tell *column* mid:get–previous)
***Returns*: previous column**
***Function 9–90:* get–parent**
(tell *column* mid:get–parent)
Returns*: Page containing *column
***Function 9–91:* get–child**
(tell *column* mid:get–child)
Returns*: first line or floating frame in *column
***Function 9–92:* get–last–child**
(tell *column* mid:get–last–child)
Returns*: last line or floating frame in *column

The following will build a list of the text contents of the first line of every column, along with the page number of each. This could be further refined to provide the first and last words of any set of columns.

9–61 first–lines

```
(defun first-lines ()
  (let (col page page-number line line-contents (col-list nil))
    ; get first page
    (setq page (tell-class doc-page-class mid:get-first))
    ; get first column
    (setq col (tell page mid:get-child))
    ; loop
    (while col
      ; get first line
      (setq line (tell col mid:get-child))
```

```
    ; get contents
    (setq line-contents
      (tell (tell line mid:get-marker) mid:get-substring
        (tell line mid:get-marker t) t t))
    ; get page number
    (setq page-number (doc-page-number-of (doc-page-of line)))
    ; add it to the list
    (push (cons line-contents page-number) col-list)
    ; get next
    (setq col (tell col mid:get-next)))
  ; return the list
  col-list
))
```

Lines

Lines change so frequently in Interleaf 5 that they usually do not form a reliable basis for programming. Nevertheless, if you need to know about them, Interleaf Lisp lets you find out all about them.

***Function 9–93:* doc-point-line**

(doc-point-line)

***Returns*: line the text caret is in**

Lines are children of columns, so you can get the first in a column by doing a *get-child* of a column.

The following will give the first line or floating frame of a column on the current page.

9–62 get-first-line

```
(defun get-first-line ()
  (let (page col)
    ; get first page
    (setq page (tell-class doc-page-class mid:get-first))
    ; get first column of page
    (setq col (tell page mid:get-child))
    ; get first line
    (tell col mid:get-child)
))
```

We can now walk through the entire document, using the familiar *get-next*.

9–63 count-lines

```
(defun count-lines ()
; uses get-first-line
```

```
  (let (line (line-ctr 0))
    ; get first line
    (setq line (get-first-line)) (break)
    ; loop through doc
    (while line
      ; increment counter
      (inc line-ctr)
      ; get next line
      (setq line (tell line mid:get-next)))
    line-ctr
  ))
```

Function 9–94: get–marker

(tell *line* mid:get–marker *argument*)

***Returns*: marker at beginning of *line* if *argument* is nil, or at end of *line* if *argument* is t**

E.g. *(tell (doc–point–line) mid:get–marker nil)*
Returns marker at beginning of line

If you don't specify any argument, this function defaults to *nil* , returning a marker at the beginning of the line.

By combining these functions, you can, for example, automatically italicize all lines of a particular type. For example, if you wanted to italicize every comment line in programming code, you could look for lines that begin with a semi-colon (or which only have tabs or spaces before a semi-colon):

9–64 italicize–all–lines

```
(defun italicize-all-lines (trigger-chars throw-out-chars)
; italicizes lines that begin with trigger char or has any throw-out-chars
; before trigger char
  (let (line m1 m2 first-chars contents)
  ; get first line of doc ... uses function get-first-line)
  (setq line (get-first-line))
  ; loop through all lines
  (while line
    ; get text content of line
    (setq m1 (tell line mid:get-marker nil))
    (setq m2 (tell line mid:get-marker t))
    (setq contents (tell m1 mid:get-substring m2 t t))
    ; throw out chars that don't count
    (if contents
      (progn
        (setq contents (string-left-trim throw-out-chars contents))
        ; get first char after the ones you've thrown out
```

```
        (setq first-chars (substring contents 0 (string-length
        trigger-chars)))
        ; do the first chars match the trigger-chars?
        (if (string= first-chars trigger-chars)
            (progn
          ; if so, then italicize the line
            ; select the line
            (tell *text-editor* mid:select m1 m2)
            ; italicize line
            (tell *text-editor* mid:set-props :italic t)
            ; deselect the line
            (tell *text-editor* mid:deselect)))))
    (setq line (tell line mid:get-next :along :format)))
))
```

You invoke this function by giving it a string of the characters you want to trigger the action (in this case, italicizing the line) and a string containing the characters it should throw out at the beginning of a line, such as tabs and spaces; for example, *(italicize-all-lines ";" " \t")* where the second argument consists of a tab and a space. (*Note:* be careful experimenting with this because once you've changed the text properties of lines, there is no fast way through the user interface to change them back.)

You can get the following properties of lines:

:begin-new-page	t if line must start new page
:begin-new-column	t if line must start new column
:straddle-columns	t if line straddles columns
:has-revision-bar	t if line has revision bar
:baseline	Distance, in rsu's, from the top of the column to the line's baseline
:height	Height, in rsu's, of the line
:width	Width, in rsu's, of the line
:doc-font	Starting font for the text in the line

With this you could, for example, report how many lines have changed (if you are using rev bars):

9-65 count-changed-lines

```
(defun count-changed-lines ()
; Counts lines with rev bars. Uses get-first-line
  (let (line (rev-ctr 0))
    (setq line (get-first-line))
    ; loop through doc
    (while line
```

```
    ; increment counter
    (if (and (not (is-of-class line doc-frame-class)) ; frames sometimes
      are lines
        (tell line mid:get-props :has-revision-bar))
        (inc rev-ctr))
    ; get next line
    (setq line (tell line mid:get-next :along :format)))
  ; report it (using ~P to make plural of "line" when necessary)
  (stk-open (format nil "~D line~P changed" rev-ctr rev-ctr))
  ; return count
  rev-ctr
))

; try it
(count-changed-lines)
```

The following will let us do something a little more interesting with changed lines:

Function 9–95: get–top–cmpn

(tell *line* mid:get–top–cmpn)

***Returns*: top level component *line* is in**

E.g. *(tell (doc–point–line) mid:get–top–cmpn)*
Returns component that contains current line

Now, rather than simply counting the number of lines with rev bars, we could make a list recording the text contents of the line, the name of the top-level component it's in, and the page it falls on. From this list we could then generate a written report of which changes occurred on which pages. (Using Interleaf 5's revision tracking capabilities, we could build more interesting reports still, noting who made which changes when.) To generate this list, we need to add a few lines to where we increment the rev–ctr in the *count–changed–lines:*

```
; get cmpn
(setq cmpn (tell line mid:get-top-cmpn))
; get contents
(setq contents (tell (tell line mid:get-marker)
  mid:get-substring (tell line mid:get-marker t) t t))
; get page
(setq page (doc-page-number-of (doc-page-of obj)))
; build list
(push (list contents cmpn page) changed-lines-list))
```

The following function is useful when writing programming code within a document. It assumes that you are using the return key at the end of a line. It counts the number of tabs at

the beginning of the line above it and inserts the same number at the beginning of the new line.

9–66 semi–smart–indent

```
(defun semi-smart-indent ()
; indents same amount as previous line
 (let (m1 m2 line start tab-ctr tabs)
  ; only works in cmpns named "code"
  (if (string= "code" (tell (doc-point-cmpn) mid:get-name))
     (progn
      ; get marker at beginning of line
      (setq m1 (tell (doc-point-line) mid:get-marker))
      (setq m2 (tell (doc-point-line) mid:get-marker t))
      (setq line (tell m1 mid:get-substring m2 't t))
      ; count tabs
      (setq start 0)
      (setq tab-ctr 0)
      (if line
        (progn
          (while (string= "\t" (substring line start 1))
            (inc start)
            (inc tab-ctr))))
      ; do the return
      (key-insert-return)
      ; build tab string
      (setq tabs nil)
      (repeat tab-ctr
        (setq tabs (concat "\t" tabs )))
      ; insert tabs
      (if tabs
        (tell (doc-point-marker) mid:insert tabs))
      ; move to end of line
      (doc-goto-marker (tell (doc-point-line) mid:get-marker t)))
    ; else
    (key-insert-return))
))

(kbd-bind kbd-doc-map "\r" 'semi-smart-indent)
```

Binding the return key to the smart–indent function is not the preferred way of operating because it means that in every document, every time you hit the return key, it's going to check the name of the component. It'd be better to make a special text editor sub-class that knows what it should do when it receives a return key. But we aren't quite up to that yet.

Function 9–96: get–bounds

(tell *line* mid:get–bounds)

Returns*: left–offset, top–offset, width, height and baseline of *line

E.g. *(tell (doc–point–line) mid:get–bounds)*
Returns (75 207 256 12 216)

Here are the returns of two identical lines separated only by a few lines:

(tell (doc–point–line) mid:get–bounds)
Returns (86 226 210 12 271)
(tell (doc–point–line) mid:get–bounds)
Returns (86 296 210 12 305)

Providing new editors

One of the most powerful capabilities of Interleaf 5 is that it allows you to create your own editor objects and attach them to a document or a class of documents. That means you can start with the default text editor, for example, modify it, and make your customized text editor the default for that document.

To do this, you first create a new class of editor. Then you give that new class whatever new methods you want it to have. Then you tell the editor of a particular document to become an editor of that class.

Here is an example:

```
; create a new class of text editor named "new–editor"
(setq new–text–editor–class  (obj–new–class doc–text–editor–class
"new–editor"))
Returns object class
; give new class a new cut method (defined elsewhere)
(obj–provide new–text–editor–class mid:cut 'new–cut–function)
Returns new–cut–function
; tell current text–editor to become a new–text–editor class of text–editor
(tell *text–editor* mid:set–class new–text–editor–class)
Returns object class
```

If you were to evaluate these lines, the text editor in your current document would look to a function called "new–cut–function" whenever the user tried to cut selected text. But, of course, since that new cut function hasn't been defined, the user would get an error message. So let's define a new cut method.

9–67 new–cut–function

```
(defun new–cut–function (editor)
  (let (old–method)
    ; get the old cut method
```

```
    (setq old-method
      (obj-get-method (tell new-text-editor-class mid:get-parent)
      mid:cut))
    ; add something new - a stickup
    (if (stk-open "Are you sure you want to cut this text?" :yes-no)
      (progn
        (funcall old-method editor)))
))
```

Let's look more closely at the *new–cut–function* function. We want our new cut function to do something new (to take another example, you might want to have all cut information automatically pasted into some other file) and then to do the normal sort of selected-text cut. So suppose you write *new–cut–function* so that it does its new bit of business and then does a *(tell *text-editor* mid:cut).* It will then go to *new–cut–function* which will tell it to do its cut method, which will send it to *new–cut–function ... ad infinitum.* So, how do you tell the new text editor to do a normal sort of cut in its new cut function?

You may have given your new text editor a new cut method, but the normal text editor still remembers how to cut the old fashioned way. Because Interleaf 5 is object-oriented and uses inheritance, when you make a new class, you do so by modelling it on an existing class. Our new–text–editor–class is based on the doc–text–editor–class that comes defined with the system. So, your new text editor cut method will tell the parent of the new text editor class to hand over its cut method (and assign it the name *old-method*). Then, after doing whatever new things we want our new text editor cut method to do, we'll execute the old cut method by saying *(funcall old-method editor)* (where "editor" is the current text-editor). (*Funcall* is short for "function call," and is not a practical joke played on the telephone.)

The last thing you need to know to make practical use of this capability is how to give a document a new text editor. You'll learn more about this in Chapter 18 on creating active documents, but basically the process is this:

- Create a comma 5 file that creates a new class of object, which defines the current document as a member of that class, and which gives that class a new open method found in some other Lisp file attached to the document.
- In the other Lisp document, write an open method that captures the open method of the parent class of the current document. The new open method should do whatever else you want this document to do on opening. Then do a *funcall* using the old open method so that the document will open in the normal way.

If you really want to get yourself in trouble, you can do the thing that Interleaf, Inc. warns you against doing: you can change the methods of the parent class of editors. The reason to do that is to change the way text editors work in all your documents, rather than having to make a new class of text editor and apply it whenever you open a document. To do this, you do something like:

```
(obj-provide doc-text-editor-class mid:cut 'new-cut-function)
```

Now every document you open will inherit the new cut method. The best way to get the normal method back is to exit the desktop and start the system up again. Note: If you've already created a text editor for a particular document that uses *new-cut-function* and you *eval* the line above, then that new text editor will look to its parent for the traditional cut function but won't find it. The result is an infinite loop! Believe me. I found out the hard way. More than once.

Here's an example that avoids the infinite loop. You know that when you do a normal *join* of two components, Interleaf doesn't presume to insert a space between the two newly-joined paragraphs. You might prefer to have the space inserted automatically. The following redefines *all* your component editors so that the space is inserted.

```
; get the old join method — make it a global
(setq *old-join-method*
  (obj-get-method doc-cmpn-editor-class mid:join))
; give cmpn editor a new join method
(obj-provide doc-cmpn-editor-class mid:join 'new-join-function)
```

9-68 new-join-function

```
(defun new-join-function (editor)
  (let (old-method)
    ; do the old  style join
    (funcall *old-join-method* editor)
    ; insert the space (even when the two cmpns are empty ... a chance to
enhance!)
    (tell (doc-point-marker) mid:insert " ")
))
```

Chapter 10
Stickups and Stayups

A stickup is a type of dialogue box that provides an important way Interleaf 5 interfaces with end users. The chief distinguishing characteristic of stickups is that they don't allow the user to do any other work until she takes care of the stickup (perhaps simply by hitting the cancel button).

As with most of Interleaf Lisp's control over the interface, you can either build a stickup object and then open it up for interaction whenever you need it, or you can build it on the fly, get the result, and no longer have it available as an object. For example, the warning stickup that is built into interleaf 5 to remind you that you're about to revert a document to its previous version contains a lot of text and is always the same. The stickup is an object that gets called repeatedly. If you have the Developers Toolkit, you can access that object, build your own stickup object, and replace Interleaf's with your own. On the other hand, if you're building an application that uses a stickup once (maybe it asks for a password) or that has to be built dynamically (perhaps it's giving choices based on the content of a particular component), then you'll build it on the fly.

We will begin by looking at how to build stickups on the fly because they give more immediate feedback as you're learning how to build them. The two approaches are very similar in any case.

Stickups

We have used stickups in earlier chapters, so the basic form is probably already familiar to you:

Function 10–1: stk–open

(stk–open *text*)

Displays *text* and a continue button

E.g. *(stk–open "Don't forget to save your work")*
Creates stickup with text "Don't forget to save your work" and returns 0 when user clicks on continue button

This function, however, can take keywords, as follows:

Keyword	Function	Return
:yes–no	Makes Yes & No buttons	t for yes, nil for no
:confirm–cancel	Makes Confirm & Cancel buttons	0 or 1
:buttons	Takes list and makes button for each	number of button (zero-based)
:left–justified	Left-aligns the text	
:font	Choice of :system or :lisp	
:interactive–choice	Specifies which is on by default	

You can create a multi-line message by listing each line as a separate string. For example:

```
(stk-open "This is line one" "This is line two")
```

Or, you can use the format function to build strings for a stickup either by using it explicitly or by specifying a list for your text. For example, the following three expressions result in the same stickup text:

```
(setq name "Pat")
(stk-open "Hi, Pat")
(stk-open (format nil "Hi, ~A" name))
(stk-open (list "Hi, ~A" name))
```

The following multi–line stickups are also equivalent:

```
; #1
(stk-open "You owed $10.65"
 "You paid $5.45"
"You still owe $5.20")
; #2
; set some variables
(setq owed 10.65)
(setq paid 5.45)
(stk-open (concat "You owed $" (ftoa owed) "\nYou paid $" (ftoa paid)
"\nYou still owe $" (ftoa (- owed paid))))
; #3
(stk-open (format nil "You owed $~,2F\nYou paid $~,2F\nYou still owe
$~,2F" owed paid (- owed paid)))
```

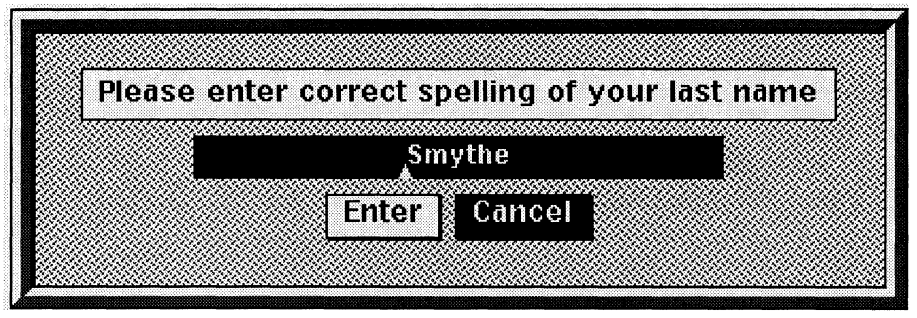
Among the advantages of using *format* over *concat* is that *concat* relies upon *ftoa* to convert the result of the calculation of the amount owed into a printable representation. But *concat* isn't smart enough to add the zero at the end of the calculation, so it prints "5.2" rather than "5.20". *Format*, on the other hand, is good at such things.

Function 10–2: stk–open–prompt

(stk–open–prompt text &optional :font font :width width :left–justify value :initial–prompt string)

Creates stickup with text and lets the user enter text and returns the text

E.g. *(setq lastname "Smythe")*
(stk–open–prompt "Please enter correct spelling of your last name" :width 30 :left–justify t :initial–prompt lastname)
Creates the following stickup:



Stickup with some text in place as a prompt.

For example, you could use a series of stickups to build the buttons for another stickup:

10–1 build–buttons–stickup

```
(defun build–buttons–stickup ()
(let ((button–list nil) button message choice)
  ; get input until user leaves it blank
  (while (not (string= ""
    (setq button (stk–open–prompt "Enter string for button" :width 10))))
    (push button button–list))
  ; reverse the order of the list
  (setq button–list (reverse button–list))
  ; get the message for the stickup
  (setq message (stk–open–prompt "Enter message you want for stickup"
:width 60))
  ; open the stickup only if user has entered right info
  (if (and message button–list)
    (setq choice (stk–open message :buttons button–list)))
choice
))
```

Interleaf Lisp provides some input stickups that are a little more intelligent than *stk–open–prompt.*

Function 10–3: stk–get–number

(stk–get–number text &optional width)

Creates stickup with text message that accepts width number of digits and returns the number

E.g. *(stk–get–number "Enter left margin" 4)*
Returns number entered.

This function returns a number rather than a string. If the user enters a string, the user is given an error message.

If you want to control what happens in case of error, you can use a more cumbersome function:

10–2 stk–open–get–number

```
(defun stk–open–get–number (message)
(let (input–string input–number)
  ; get the input as text
  (setq input–string (stk–open–prompt message :width 4))
  ; if anything entered
  (if input–string
    ; try to convert to float
    (setq input–number (atof input–string)))
  ; if couldn't convert to float, give an error message
  (if (not input–number)
    (stk–open "Input was not an acceptable number"))
input–number
))
```

You might alter that function so that it checks that the number is within a particular range. Or you might change it so it loops until the user either provides acceptable input or hits the cancel button. In this latter case, you might want to alter the stickup's message to reflect the fact that the previous input failed. For example:

10–3 stk–open–get–number–loop

```
(defun stk–open–get–number–loop (text)
(let (input–string (input–number nil) (input–failed nil))
  (while (not input–number)
    ; get the input as text
    (setq input–string (stk–open–prompt text :width 4))
    ; try to convert to float
    (if (not (setq input–number (atof input–string)))
      (stk–open (concat input–string " not acceptable. Input a number."))))
input–number
))
```

Function 10–4: stk–parse–numeric

(stk–parse–numeric *text default* &optional *arguments*)

Creates stickup with message *text* and returns *default* if the user enters no value or returns the value of the text entered.

The point of this function is that it allows the user to enter not only a number but a mathematical expression. The function will do the calculation and return the amount. For example, if the user entered the text "2 * (100 / 4)" the function will return the number 50.0. (Notice it returns a float.)

The optional arguments allow you to use the publishing–specific knowledge built into the system: *:input–units*, *:output–units* and *:doc–font.* The first two allow you to specify *:inches*, *:mm*, *:picas*, *:points*, *:didots*, or *:ciceros.*

For example:

```
(stk–parse–numeric "Enter inches" 0 :input–units :inches :output–units
:mm)
```

Returns a number that has converted the number entered into millimeters. For example, an input of "1.5" would return 38.1, which is the millimeter equivalent of 1.5 inches.

The *:doc–font* argument allows you to specify which font you want used when parsing expressions such as "line." For example, using the following you could enter "1 line" and have it parsed to the height of one line in the specified font:

```
(stk–parse–numeric "Test" 0 :doc–font (tell (doc–point–cmpn)
mid:get–props :doc–font))
```

Now let's look at creating stickups as objects and then invoking them whenever you need them. It's simple.

Function 10–5: stk–new–choice

(stk–new–choice &optional :text :buttons :interactive–choice :font :left–justify)

Returns number of button that was chosen

E.g. *(stk–new–choice :text "Choose your font" :buttons (list "Swiss" "Helv" "Thames")*
:left–justify t)
Returns stickup object

No stickup appears on screen when you use this function. To invoke the stickup you use the following function

Function 10–6: stk–up open

(tell *stkup–obj* mid:open)

Creates stickup and gives the result of the user interaction with it

E.g. *(tell font–stkup mid:open)*
Returns result of user interaction with font–stkup

This works with one other type of stickup object you can create:

Function 10–7: stk–new–prompt

(stk–new–prompt &optional :text :buttons :interactive–choice :font :left–justify)

Returns stickup object

This works along the same lines as *(stk–new–choice)*.

Stayups

When a stickup is on screen, the user is blocked from doing anything until she or he has interacted with the stickup. Sometimes, however, you want to put information up on the screen without blocking other actions. Stayups are one way of doing this.

A stayup is, basically, a plain text window; all it can do is display characters. Why use it instead of a document? Stayups open faster and offer less distracting functionality than docuents do. If you need multi-font display, or graphics, then you might want to use a document, but open it hidden *(tell document mid:open :hidden t)* when your application opens, and then open it in the normal way when you want to display it; this puts the overhead of opening the document into the load time of your application.

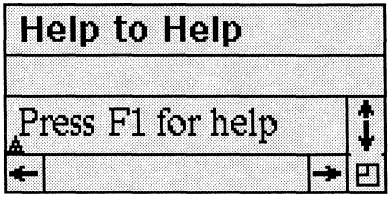
Opening a stayup is as easy as opening a stickup:

Function 10–8: stayup

(stayup *message* &optional :title *title* :font *font* :position *pos* :size *size*)

Creates stayup with contents of *message*, title of *title*, in font *font*, as position indicated by the cons *pos* and of size indicated by the cons *size*. Returns the window object.

E.g. *(stayup "Press F1 for help" :title "Help to Help" :font "thames14" :position (cons 100 200) :size (cons 150 75))*
Creates stayup illustrated below, 100 pixels to the left and 200 pixels down.
It is 150 pixels wide and 75 pixelsels high.



A stayup.

If you don't specify a position, the stayup is positioned at the cursor position. If you don't specify a size, the stayup sizes itself to fit the text.

You specify the font by using the internal name of a recognized font. (You can get a list of those names through *(fn–families)*. Remember to put it in lowercase with the type size appended to it, as in the example above. The default font Interleaf 5 uses in stayups is "lisp10."

(fn–families)
Returns (("SysErrors" "SysMarkers") ("Courier" "Helvetica" "Swiss" "Thames" "Times") ("Century" "HelveticaNarrow") ("ThamesGreek") ("Math A" "Math B" "Math Ext." "Math–8502") ("Symbols" "ZapfDingbats") ("Symbol_A" "Symbol_B"))

Function 10–9: stayup–close

(stayup:stayup–close &optional window)

Closes the latest stayup or window

E.g. *(setq test–stayup1 (stayup "This is a test"))*
(setq test–stayup2 (stayup "This is a second test"))
(stayup:stayup–close test–stayup1)
(stayup:stayup–close test–stayup2)
Creates two stayups, closes the first one, then closes the second one

Notice that *stayup–close* requires the package name before it, as shown in the example. (Packages are not within the scope of this book.)

If you have lost track of your stayups, the following function will return the stayup window based on the titles. (If the stayup doesn't have a title, it won't work. Also, if there is more than one window with the same title, it will return only one of them.)

10–4 find–stayup–window

```
(defun find–stayup–window (title)
; finds a stayup window with name "title"
(let (obj win–list window win–title (found–win nil))
  ; get list of windows
  (setq win–list (tell *wn–wmgr* mid:get–props :windows))
  ; look through list for title
  (while (setq window (pop win–list))
    (if (and
      (setq obj (tell window mid:get–object))
      (is–of–class obj stayup:stayup–class))
        (progn
          ; get title
          (setq win–title (tell obj mid:get–name))
          (if (and win–title
            (string= win–title title))
              (setq found–win window)))))
found–win
))
```

You can update the contents of a stayup in either of two ways.

Function 10–10: optional stayup:add–string

(stayup:add–string *text window* &optional :refresh)

Adds *text* to window and returns the *string.* If *:refresh* is t, immediately updates the window.

E.g. *(setq status–stayup (stayup "Found file ..."))*
(stayup:add–string "Processed file" status–stayup :refresh t)
Creates stayup, adds line, and returns "Processed file")

If you don't make *:refresh* t, then the window is updated whenever its time comes.

Function 10–11: optional stayup:add–line

(stayup:add–line *text window* &optional :refresh)

Adds *text* to *window* and adds new line; if *:refresh* is non–nil then immediately updates window

This works exactly the same as *stayup:add–string* except it puts in a carriage return at the end of the line.

The following function will put the pathname of a desktop object into a stayup:

10–5 pathname–stayup

```
(stayup (dt–get–property (dt–child–selected) :path–name))
```

This is especially useful in DOS where the desktop name is frequently very unlike its operating system name.

The following set of functions takes any selected text and puts it into a stayup. This is useful when you want to "pin up" some text from one page in a document while you work elsewhere in the document. There are two challenging parts of this: it's harder than it seems to get text when the selection spans components, and we need to recompute the line endings to fit into our stayup. Here's the code:

```
; create a variable used to specify how many characters on a line
(setq *st–fmt* 30)
```

10–6 text–into–stayup

```
(defun text–into–stayup ()
; finds selected text and builds stayup
(let (this–stayup m1 m2 text c1 c2 (new–text "") done this–cmpn)
  ; if nothing is selected, quit
  (if (not (doc–text–selection))
    (quit))
  ; create the stayup
  (setq this–stayup  (stayup text :font "swiss10"
       :title (format nil "~A, pg. ~D" (tell *document* mid:get–name)
```

```
        (tell *doc-editor* mid:get-props :page))))
  ; get markers at beginning and end of selection
  (setq m1 (car (doc-text-selection)))
  (setq m2 (cdr (doc-text-selection)))
  ; try to get the text ... only works if it's all within one cmpn
  (if (setq text (tell m1 mid:get-substring m2 t t))
    (setq done t))
  ; get the cmpns the beginning and end markers are in
  (setq c1 (tell m1 mid:get-parent))
  (setq c2 (tell m2 mid:get-parent))
  ; if multi-cmpn, get first cmpn's text
  (if (not done)
    (setq text (tell m1 mid:get-substring (tell c1 mid:get-marker t) t t)))
    (if text
      (progn    ; format the text and stick it into the stayup
        (setq text (format-text text))
        (put-text-into-stayup text this-stayup)))
  ; loop through cmpns until you're in the one that end marker is in
  (setq this-cmpn c1)
  (while (not done)
    ; get the next cmpn
    (setq this-cmpn (tell this-cmpn mid:get-next))
    ; is it the same as the end marker's cmpn
    (if (not (eql this-cmpn c2))
      (progn    ; get the text
        (setq new-text
          (tell (tell this-cmpn mid:get-marker) mid:get-substring
          (tell this-cmpn mid:get-marker t) t t)))
        ; else, if this is c2, get rest of text
        (progn
          (setq new-text
            (tell (tell this-cmpn mid:get-marker) mid:get-substring m2 t t))
          (setq done t)))
      ; format the text and stick it into the stayup
      (setq text (format-text new-text))
      (put-text-into-stayup text this-stayup)
)
))
```

10–7 put–text–into–stayup

```
(defun put-text-into-stayup (text this-stayup)
; puts text into stayup and adds 2 linefeeds
  (stayup:add-string (format nil "~A\n\n" text) this-stayup)
)
```

10–8 format–text

```
(defun format-text (text)
; does word wrap for text at line length specified by *st-fmt*
(let (done new-pos m1 m2 new prev prev-sp m
  (pos 0) (space-list nil) (eol-list nil) (newtext "") (len *st-fmt*))
  ; create list of where the spaces are
  (while (not done)
    (if (setq new-pos (string-contained " " text pos))
      (progn
        (push new-pos space-list)
        (setq pos (1+ new-pos)))
      ; else
      (setq done t)))
  ; space list is in reverse order, so turn it around
  (setq space-list (reverse space-list))
  ; set a bunch of counters to 0
  (setq m 0 prev 0 prev-sp 0)
  ; create list of where the breaks ought to be
  (while (setq new (pop space-list))
    ; look for the difference between last place we started a line and next
    ; spot on list of spaces
    ; if difference is greater than margin, then note this spot
    (if (>= (- new prev-sp) len)
      (progn
        (push prev eol-list)
        (setq prev-sp prev)
        ;(push new space-list)
      ))
  (setq prev new))
  ; snip together the string
  (setq eol-list (reverse eol-list))
  (setq m1 0)
  (while (setq m2 (pop eol-list))
    (setq newtext (concat newtext 10 (substring text m1 (- m2 m1))))
    (setq m1 (1+ m2))
  )
  (setq newtext (concat newtext 10 (substring text m1 (- (string-length
```

```
  text) m1))))
newtext
))

; bind it to ^XN
(kbd-bind kbd-doc-map "\^XN" 'text-into-stayup )
```

Chapter 11
Popups

Popups are obviously a key part of Interleaf's standard interface. In this chapter we'll show how to create a popup object and how to get that object attached to various objects so it can actually be used.

Creating popups

A popup consists of a list, each item of which corresponds to one entry on the popup menu. Let's begin by making a single entry.

Function 11–1: popup–new–entry

(popup–new–entry *text &optional* :handler *function* :sublist *popup–list* :data *data*)

***Returns*: popup entry for use in popup object**

E.g. *(popup–new–entry "Set props"*
:handler 'my–set–props
:sublist prop–sublist–pup
*:data (tell *document* mid:get–name))*
Returns popup entry with text "Set props" which will run 'my–set–props when selected, which has as a sublist another popup entry called prop–sublist–pup, and which passes to the handler the name of the document

Every popup entry you create will have at least some text to appear on the menu and a handler, i.e., a function invoked when the user chooses the popup. If you want it to have a sublist, then give it another popup after the *:sublist* key word. And there are times when the handler function could use some additional data; the *:data* keyword lets you append whatever type of data you want.

Remember, this function by itself will not cause a popup to appear. To do that we first have to build a complete popup.

Function 11–2: popup–new–list

(popup–new–list *popup–list &optional* :offset :font :default)

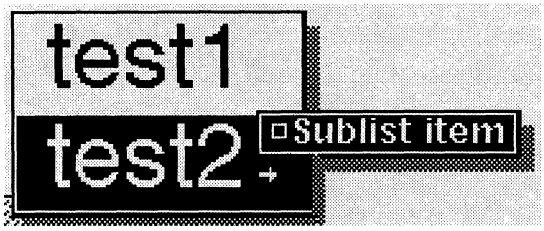
Returns*: popup object consisting of entries on *popup–list

This function takes a list of popup entries and turns them into a popup object. The keyword *:offset* can be either *:right* or *:down;* the former is the normal type of popup, and the latter is the pulldown sort of popup you get in the document header bar. The font is expressed in the form of "swiss36," "thames24," etc. (The default font, at least on my Sun SPARC, is "wst:Ops14.") The :default keyword is inserted into the list of entries to force one particular entry to be the default; otherwise, the system will choose the center one as default (or the higher of the two center ones if there are an even number of entries).

The easiest way to understand this is to look at an example.

11–1 popup–example

```
; create entry to be used as a sublist entry later
(setq test–sublist
  (popup–new–list  ; create list with only one entry
    (list
      ; create an entry that will execute a function called "sublist–function"
      ; (which we have not defined)
      (popup–new–entry "Sublist item" :handler 'sublist–function))))

; create the popup that contains the sublist item above
(setq test–pup
  ; create the popup list
  (popup–new–list
    (list
      ; create first entry, with a handler that just opens a stickup
      (popup–new–entry "test1" :handler '(stk–open "test1"))
      ; create second entry, with sublist, and make this entry the default
      :default (popup–new–entry "test2" :handler '(stk–open "test2")
:sublist test–sublist)
    )

  :font "swiss36"  ; force the font of the list to be Swiss 36
  :offset :down  ; force it to be a pulldown sort of popup
  ))
  ; make it run
(popup–run test–pup)
```



Result of test popup. Notice sublist is normal font because we didn't specify it should be different.

Function 11–3: popup–run

(popup–run *popup–object* &optional :position *(cons x y)*)

E.g. *(popup–run test–pup :position (cons 0 0))*
Runs the popup object "test–pup"and positions at upper left of screen

Now at last you can run your popup. If you don't specify a *:position,* the popup will appear at the cursor position, as usually happens.

Function 11–4: popup–get–prop

(popup–get–prop *popup–object* :prop *:property)*

***Returns*: property of popup list or popup entry**

E.g. *(popup–get–prop test–sublist :list)*
Returns list of popup entries for test–sublist

You can give this function either a popup object or a particular entry in one of those objects. If you give it a popup object, the properties you can ask for are *:offset, :font, :list,* or *:default.* The property *:list* returns the actual list of popup entries. If you have not specified a specific property when building the popup entry, this function returns nil.

If you specify a popup entry, the properties are *:name* (the text shown in the popup), *:handler, :sublist, :data, :has–data* (t if *:data* was specified when popup entry was built), and *:visible* (t if entry is visible, and not hidden).

Function 11–5: popup–decode–list

(popup–decode–list *popup–list*)

***Returns*: list of data about the popup list**

Function 11–6: popup–decode–entry

(popup–decode–list *popup–entry*)

***Returns*: list of data about the popup entry**

The easiest way to understand these functions is to look at examples. We'll use the test–pup we created above.

```
(popup–decode–list test–pup)
```

```
Returns list:
(
   ("test1" (stk-open "test1") nil nil)
   ; note sublist because of subentry
   ("test2"      (stk-open "test2")
                 (
                 ("Sublist item" sublist-function nil nil)) nil)
)
```

The information is in the order: *:name, :handler, :sublist, :data.*

*Function 11–7: *popup-history**

Returns: history of the most recently-popped popup

E.g. **popup-history**
Returns ((#<popup-list 0x71ea58> . #<popup-entry 0x71ea80>)
(#<popup-list 0x71eaa8> . #<popup-entry 0x7bef18>)
(#<popup-list 0x7bef40> . #<popup-entry 0x7bef90>))

This system variable represents the most recently used popup. You could then use the decode functions to get the particulars about this popup.

*Function 11–8: *popup-action**

popup-action

Returns: list with data about most recently used popup

E.g. **popup-action**
Returns ((:handler . ileaf::text-editor-ph) (:data #<mid:paste>)
(:context))

This function returns three values: the handler the popup invokes, the data that gets passed to the handler, and any context information that may have been passed to the handler.

*Function 11–9: *popup-last**

popup-last

Returns: the last popup entry used and visible

E.g. **popup-last**
Returns #<popup-entry 0x7b0340>

The trick here is that if you use a popup but click through it so quickly that it doesn't have time to display itself (e.g., if you're a power user and after doing a cut you just do a click to perform the paste), the value of **popup-last** will be nil.

Attaching popups to objects

There are two fundamental ways to get your popups to run. First, you can substitute your popup for one of the existing ones and let the editor objects automatically use yours. Second, you can hook into an event and write the code to get your popup to run.

To substitute yours for an existing editor's, you create a new class of text editor and alter that editor's popup method. For example:

11–2 change–editor–popup

```
; create a new subclass of text editor
(setq new–text–ed (obj–new–class doc–text–editor–class "new text
ed"))
; give that new class a new popup method
(obj–provide new–text–ed mid:popup 'new–popup–fcn)
; reclass this text editor to be new–text–ed class
(tell *text–editor* mid:set–class new–text–ed)
```

Now when you press the mouse menu button, you'll get an error message since you have not created *new-popup-fcn* which is what pressing the menu button looks for. So, let's create the function we need:

11–3 new–popup–fcn

```
(defun new–popup–fcn (name handler data)
  ; run a previously–created popup
  (popup–run test–pup))
```

The same sort of procedure can be used on the component editor and the table editor.

This technique provides you with the same popup wherever the editor is in force. But suppose you want the user to get different popups in different components. Here is some code that may help.

11–4 context-sensitive-popups

```
; create a new subclass of text editor
(setq new–text–ed (obj–new–class doc–text–editor–class "new text
ed"))
; give that new class a new popup method
(obj–provide new–text–ed mid:popup 'new–popup–fcn)
; reclass current text editor as of new–text–ed class
(tell *text–editor* mid:set–class new–text–ed)
```

11–5 new-popup-function

```
(defun new–popup–fcn (&optional text handler data )
; looks for attribute called "popup"
(let (attr symb pup)
  ; is there an attribute?
  (if (setq attr (tell (doc–point–cmpn) mid:get–attrs "popup"))
    (progn
      ; if a "popup" attribute, convert value to symbol
```

```
        (setq symb (eval (find-symbol attr)))
        ; if there's a popup with that name, run it
        (if symb
          (setq pup (eval (find-symbol attr))))))
      ; else no attribute specified, so run the normal popup
      (if (not pup)
        (progn
          ; if text is selected, then run normal text-selected popup
          (if (doc-text-selection)
              (setq pup ileaf::doc-text-sel-pl)
              ; else, nothing is selected, so run normal no-sel popup
              (setq pup ileaf::doc-text-pl))))
    (popup-run pup)
  ))
```

Notice that this uses a function we don't otherwise discuss in this book: *find-symbol* takes text and looks for a symbol with that name; *eval*'ing the return gives you the symbol itself. Also, this function sets the popup to the already-created standard text popups if no popup is specified in the popup attribute; this is explained below.

Altering existing popups

In addition to making your own, you can get and alter the popups that come with the system. To get an existing popup you have to know its symbol name. Those names, by and large, are defined in the lisp directory in your Interleaf directory. Here are some of the key popup names:

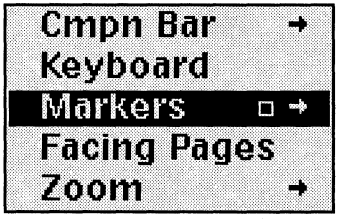
Popup name	Comment
Text	
doc-text-pl	main text popup, nothing selected
doc-text-sel-pl	main text popup, something selected
text-read-only-sel-pl	if in read-only text
text-read-only-nosel-pl	if in read-only text
doc-text-misc-pl	sublist of misc submenu
Graphics	
dg-norm-nosel-pl	main popup, nothing selected
dg-norm-sel-pl	main popup, something selected
dg-select-pl	selection submenu
dg-edit-pl	edit submenu
dg-norm-nosel-misc-pl	misc submenu, nothing selected
dg-sel-misc-pl	misc submenu, something selected
dg-edge-dashes-pl	dashes

dg–fill–pl	fills
dg–edge–pl	edges
dg–edge–weight–pl	edge-weight
dg–edit–font–pl	edit font
dg–create–pl	create diagramming objects
dg–create–subedit–pl	go to subedit
dg–create–arc–pl	create arc
dg–create–old–text–pl	create old style text
dg–view–magnify–pl	view magnified
dg–size–pl	size submenu
dg–size–to–frame–pl	size to frame
dg–size–numeric–pl	size numeric
dg–move–rect–pl	constrained move
dg–move–numeric–pl	numeric move
dg–rotate–pl	rotate
dg–dup–pl	duplicate
dg–shear–pl	shear
dg–align–pl	align submenu
dg–align–to–frame–pl	align to frame
dg–convert–pl	convert submenu
dg–convert–to–ortho–pl	convert to ortho view
dg–convert–to–iso–pl	convert to isometric view
Components	
doc-cmpn-sel-pl	main menu, something selected
doc-cmpn-pl	main menu, nothing selected
Tables	
row–nosel–pl	main row menu, nothing selected
row–sel–pl	main row menu, something selected
col–nosel–pl	column not selected
col–sel–pl	column selected
cell–sel–pl	cell selected
Document header boxes	
doc–name–box–pl	main pulldown in name box in header
doc–name–box–save–pl	save submenu
doc–name–box–revert–pl	revert submenu
doc–name–box–read–only–pl	pulldown if document is read-only
doc–page–box–pl	main pulldown in page box
doc–print–box–pl	main pulldown in print box
doc-view-box-pl	main pulldown in view box

doc–view–box–cmpn–pl	cmpn submenu in view box
Desktop	
dt–nosel–pl	main desktop menu, nothing selected
dt–sel–pl	main desktop menu, something selected
dt–cut–or–purge–pl	cut or purge
clipboard–nosel–pl	clipboard menu, nothing selected

To use one of these, you must put ileaf:: in front of it; this tells it which package to look in. (Packages are an advanced topic this book does not discuss.) For example:

```
(popup–run ileaf::doc–view–box–pl)
Returns popup in view box
in document header
```



Pulldown off View entry in View box in document header.

Interleaf, Inc. explicitly warns against altering the default popups that come with the system because you can never tell what other code may depend upon those popups being in their original state. To avoid those problems, if you want a document that has a modified default popup, you should make a copy of the default popup, alter it, make a new class of document, and attach your altered popup to it.

This function will create a copy of a popup:

11–6 copy-popup

```
(defun copy–popup (old–pup)
(let (decoded–pup)
  ; decode the original popup
  (setq decoded–pup (popup–decode old–pup))
  ; create new popup out of list form of original
  (popup–make–list decoded–pup)
))

; try it
(setq new–pup (copy–popup ileaf::doc–name–box–pl))
```

Now that you have created a new copy of your default popup, you can insert a new entry.

Function 11–10: popup–insert

(popup–insert *popup–object popup–entry*)

Inserts *popup–entry* into *popup–object*)

E.g. *(popup–insert new–pup (popup–entry "New choice" 'new-choice-handler))*
Creates popup entry with "New choice" as text and inserts it into new–pup

You can tell a popup where you want the new entry inserted by supplying an additional argument. An argument of t will insert the popup at the end of the popup. An argument that is a particular popup entry will insert the new entry after it. If you don't supply any argument, the new entry gets inserted at the top of the popup.

The following example creates a new copy of the component bar popup when something is selected, and adds a function called "Delete contents" after the "Cut" option.

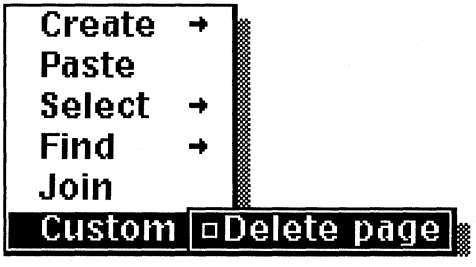
11–7 find-pup-entry

```
(defun find–pup–entry (pup text)
; finds an entry in a popup by searching for its text
(let (decoded–pup pos entry)
  ; turn popup into a list we can look at
  (setq decoded–pup (popup–decode pup))
  ; find entry in the list
  (setq entry (assoc text decoded–pup 'string=))
  ; find position of entry in decoded list
  (setq pos (position entry decoded–pup))
  ; get the entry
  (if pos
    (nth pos (popup–get–props pup :list)))
))

; create copy of cmpn popup, using function above
(setq new–cmpn–pup (copy–popup ileaf::doc–cmpn–sel–pl))
; create new entry (with a handler we haven't really created yet!)
(setq new–entry (popup–new–entry "Delete contents" :handler
'del–cmpn–contents))
; find "Cut" popup entry in the new popup
(setq cut–entry (find–pup–entry new–cmpn–pup "Cut"))
; insert the new entry after "Cut"
(popup–insert new–cmpn–pup new–entry cut–entry)
; try running the new popup
(popup–run new–cmpn–pup)
```

There is a mechanism built into Interleaf 5 that allows you to add entries to several important popups. Suppose you make a component editor called *new–ed.* You can now give it a new *get-custom* method:

```
; create new component editor
(setq new–ed (obj–new–class doc–cmpn–editor–class "sample cmpn
ed"))
; give new–ed a get–custom method called new–ed–pup–method
(obj–provide new–ed mid:get–custom 'new–ed–pup–method)

; create the new–ed–pup–method
(defun new–ed–pup–method (a b)
  ; what popup to display?
  new–ed–pup
)

; create the new popup
(setq new–ed–pup
  (popup–new–list
    (list
      (popup–new–entry "Delete page" :handler 'del–page))))
```

This will hang a popup off a Custom entry at the bottom of the component bar menu, as shown in the screen capture.



Custom entry on text popup.

Popup variables

You can set some variables that will affect all popups throughout your system.

Argument	Type	Comment
:moresize	Integer 5–256	Number of entries before you get a "More" entry and the popup splits
:quick-clicks-effective	t or nil	Lets you "beat the popup" so that a click brings the default action even before the popup displays
:timeout	Integer 0-100	The amount of time (in milliseconds) the mouse button can be down before the menu pops up
:stayup	t or nil	If t, the popups stay up even when the menu button isn't down
:font	string	Sets font. Expects string such as "swiss36" (sets font to Swiss, 36pt.)

For example,

```
(popup-set-vars :moresize 10 :timeout 50)
```

Or, another very popular example:

```
(popup-set-vars :font "swiss24")
```

This will give you nice big popups. (But beware: you may lose the arrows indicating submenus after using these font-altered popups for a while. This is an annoying bug.) By the way, if you specify an illegal font name (e.g., "Swiss24" instead of "swiss24"), the system substitutes a serif font of the appropriate size.

Chapter 12
Keyboard

Everytime you press a key on your keyboard, you're executing an Interleaf Lisp script. In this chapter, you'll learn how to alter those scripts. At the simplest level, this allows you to customize your keyboard. At the most powerful, you can radically alter your interface.

When you press a key, the keyboard sends some sort of signal to the computer. What the signal is varies with every type of computer. Interleaf knows how to interpret the signals for every computer it runs on, and you need never concern yourself with this raw level of data. Instead, what matters is what becomes of that signal once it's been received and understood by Interleaf.

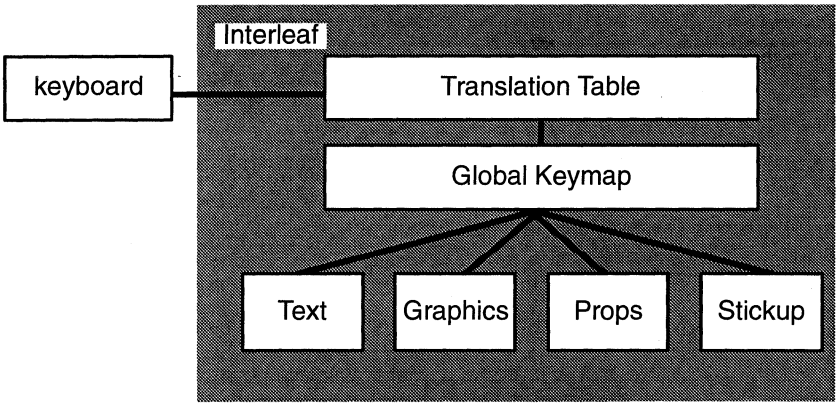
The first place the signal goes is to a *translation table*. This is an Interleaf list (different for each type of computer) that translates the signal into something that Interleaf understands (pretty much the same for each platform).

The translation table takes those interpretations and passes them to the *global keymap*. This is a list of what to do with each of the signals regardless of what context you're in — whether you're typing into text, entering a name into a property sheet, changing a name on the desktop, etc.

The information is then passed to a *local keymap*. This is also a list of translations. If a key defined in the global keymap has been redefined in the local keymap, the translation in the local keymap is the one that holds.

There are two important facts about local keymaps. First, there can be different local keymaps for text, diagramming, stickups, property sheets, etc. This means that the same keystroke can do different things depending on your context. Second, different documents can have different local keymaps so that one document might have one keyboard interface and another, a different one.

To generate a list of the local maps, look at the global variable **kbd−keymap−list**.



Relation of keymaps.

Function 12–1: *kbd–keymap–list*

kbd–keymap–list

***Returns*: list of keymaps defined**

E.g. *(*kbd–keymap–list*)*
Returns ((kbd–doc–map) (kbd–dg–map) (kbd–eq–map)
(kbd–dt–map)
(kbd–fm–map) (kbd–sr–map kbd–fm–map))

To get a complete list of the bindings of a keymap:

Function 12–2: kbd–map–list

(kbd–map–list *keymap*)

Returns list of pairs (character and definition) for all bound keys in *keymap*

E.g. *(kbd–map–list kbd–eq–map)*
Returns list of all characters and definitions in the equations map:
((#\\^X . kbd–eq–ctl–x–map) (#\\^Y . key–eq–paste)
(#\\^[. kbd–eq–esc–map) (#\\^A . key–eq–eqn–start)
(#\\^B . key–eq–prev–char) (#\\^D . key–eq–cut)
(#\\^E . key–eq–eqn–end) (#\\^F . key–eq–next–char)
(#\\f . key–eq–refresh) (#\\r . key–eq–deselect)
(#\\^N . key–eq–deselect–top) (#\\^P . key–eq–select–next)
(#\\^? . key–eq–cut–prev))

(Remember that ^X means control-X.) If you know what the key is and want to get its value in a particular keymap:

12–1 key–value–in–map

```
(defun key–value–in–map (key kmap)
(let (decoded–map)
  ; decode the map
  (setq decoded–map (kbd–map–list kmap))
```

```
  ; find key and return its function
  (cdr (assoc key decoded-map))
))

; try it out
(key-value-in-map #\\^J kbd-doc-map)
; Returns key-ec-cmpn-create
```

We will begin by looking at how to change the bindings of existing keymaps and then we'll look at creating your own keymaps and attaching them to different contexts.

To alter a key binding in an existing keymap:

Function 12–3: kbd–bind

(kbd–bind *keymap key function*)

Binds *function* to *key* on *keymap*

E.g. *(kbd–bind kbd–dt–map "\^E" 'my–function)*
Binds ^E to the function my–function on the desktop keymap

This function will accept a series of pairs of keys and functions. If instead of a key, you supply a dotted pair of keys, all the keys between (and including) those keys will be bound to the function. For example:

```
(kbd-bind kbd-dt-map (cons "\^A" "\^Z") nil)
```

This makes all the control keys do nothing.

To succeed with *kbd–bind*, though, you need to know how to express keys in strings. Simple characters are easy: you can supply the code number or just put the character in quotation marks (E.g. "A" or 65).

Function keys as well as special characters (control codes, etc.) can be expressed by prefixing them with "\^x*". For example:

"\^x*L02"	Left function key 2
"\\^x*Cut"	"Cut" function key
"\^x*F01"	Function key 1 (on DOS and others)

You can specify that a key is in its shifted, control, or meta (escape) mode by putting a number after the "*". A shift is worth 1, control 2, and meta (escape) 4. You add together the appropriate values. For example:

"\^x*1R09"	Shifted right function key 9
"\^x*2R09"	Control right function key 9
"\^x*4R09"	Meta right function key 9
"\^x*3R09"	Shifted control right function key 9
"\^x*5R09"	Shifted meta right function key 9
"\^x*6R09"	Control meta right function key 9
"\^x*7R09"	Shifted control meta right function key 9

Here are some quick functions you can bind to keystrokes:

12–2 keyboard–tricks

```
; ^Xa in graphics opens arrows clipart (put in your own pathname!)
(kbd–bind kbd–dg–map "\^Xa"
  '(tell (dt–object "System5.cab/Graphics.drw/Arrows.fdr") mid:open))

; ^Xb in graphics opens your own clipart collection named "myclipart.doc"
(kbd–bind kbd–dg–map "\^Xb"
  '(tell (dt–object "System5.cab/myclipart.doc") mid:open))

; ^Xp on desktop opens profile drawer
(kbd–bind–doc kbd–dt–map "\^Xp" '(tell *dt–profile* mid:open))

; ^Xc on desktop purges selected icon
(kbd–bind kbd–dt–map "\^Xc"
  '(progn (dt–cut–or–purge (dt–pointer–container) t)))

; ^L in a document centers page in window
(kbd–bind kbd–doc–map "\^L"
  '(let ((m (doc–point–marker)))
  (tell *doc–editor* mid:set–props :page t)
  (doc–goto–marker m)))
```

There is a slight variant of *kbd–bind*.

Function 12–4: kbd–bind–doc

(kbd–bind–doc *keymap key function doc*)

Binds function to *key* on *keymap* and uses *doc* as doc–string in keyboard prop sheet

This works precisely the same as *kbd–bind* but it has an extra argument: a brief string describing what the key does. This gets used in the visible keymap property sheet available on some systems. A typical doc string might be "smart\nquote". The "\n" gets turned into a carriage return so that this doc string will show up as two very brief lines.

There are two key prefixes that come as default; if the user types either meta (usually the escape key) or ^X, the system waits for the user to type the next key and then interprets that key as part of the escape or ^X key maps. For example, by default, ^Xs is bound to the *save–document* function in a document.

The following examples bind keys in these two maps:

```
(kbd–bind kbd–esc–map "A" 'my–function)
Binds Esc–A to my–function
```

(kbd-bind kbd-ctrlx-map "B" 'new-function))
Binds ^XB to new-function

If you want to use some other character as a prefix, you specify a keymap as the function to be invoked by a key. We'll go into this in more detail later in the chapter.

There are some very useful, simple functions available to you:

Function 12-5: kbd-self-insert
(kbd-self-insert)
Inserts the current character

This function is useful when you have bound a function to a keystroke, and the function evaluates the situation and either does something interesting or simply passes the character through. For example, the following rebinds the hard return key so that it checks the component's name to see if it's an entry blank. If it is, then it takes the user to the next component. If it's not, it simply inserts the hard return.

12-3 forms-hard-return

```
(defun forms-hard-return ()
(let (new-c (c (doc-point-cmpn)))
  ; are we in a cmpn named "blank-form"?
  (if (string= "blank-form" (tell c mid:get-name))
    (progn
      ; go to next cmpn named "blank-form"
      (setq new-c (tell *cmpn-editor* mid:goto :next "blank-form"))
      ; go to that cmpn
      (doc-goto-marker (tell new-c mid:get-marker t)))
    ; else, put in hard return
      (key-insert-return))))

; bind it to hard return
(kbd-bind kbd-doc-map "\r" 'forms-hard-return)
```

Notice that this function only works with toplevel components, not inlines. The function *(key-insert-return)* is the built-in function that gets called when you hit the enter (or return) key.

The following set of scripts does something more complex. They rebind the hard return so that it checks to see if the previous character was a hard return. If it is, then it deletes the previous return and creates a new instance of the component (as linefeed normally does).

12-4 double-return

```
(defun double-return ()
(let ((prev-char nil) (m1 (doc-point-marker)) m2)
  ; get marker for previous character
```

```
(setq m2 (tell m1 mid:copy))
(tell m2 mid:move-by -1)
; if could move back one, then get the substring
(setq prev-char (tell m2 mid:get-substring m1 t t))
; is it a return?
(if (equal "\n" prev-char)
  (progn
    ; delete old one
    (setq m1 (doc-point-marker))
    (setq m2 (tell m1 mid:copy))
    (tell m2 mid:move-by -1)
    ; select the old return and cut it
    (tell *text-editor* mid:select m1 m2)
    (tell *text-editor* mid:cut)
    ; set cmpn caret position
    (tell *cmpn-editor* mid:set-props :caret-direction :next)
    ; deselect all
    (tell *cmpn-editor* mid:deselect :all)
    ; insert new cmpn
    (tell *cmpn-editor* mid:create (tell (doc-point-cmpn)
    mid:get-name)))
  ; else
  (key-insert-return))
(doc-flush-queue)
))

; bind it
(kbd-bind kbd-doc-map "\r" 'double-return)
```

Note that this may not do exactly what you want. In particular, if you move into the middle of a component and type a double return, you may expect it to divide the component in two; that's what happens with word processors. In this case, it will instead create a new component at the end of the current one. To get it to behave the way a word processor does, instead of telling the component editor to create a new component, you could tell the text editor to split the component.

There are other functions for inserting material.

Function 12–6: kbd–insert–string

(kbd–insert–string *arg*)

Inserts string, character or character value of an integer

E.g. *(kbd–insert–string "Testing")*
Inserts the string "Testing"

Function 12–7: kbd–insert–character

(kbd–insert–character *arg*)

Inserts arg which is a character, integer or a one–character string

E.g. *(kbd–insert–character 65)*
Inserts the character "A"

Here is a function that inserts your name wherever you type ^Xn. (Of course, you can easily modify it to insert any string you want.)

12–5 insert–name

```
(defun insert–name ()
  (kbd–insert–string "Your Name"))

; bind it
(kbd–bind kbd–doc–map "^Xn" 'insert–name)
```

Now let's see how to create new maps and attach them to objects.

To get a map:

Function 12–8: kbd–get–vars

(kbd–get–vars &optional *map*)

Returns map specified by keyword *map* (:local–map or :global–map)

E.g. *(kbd–get–vars :local–map)*
Returns local key map

There are other keywords available:

:current–key returns the integer value of the current key
:current–key–str returns a string representation of current key

You can set a map in a similar way:

Function 12–9: kbd–make–sparse–keymap

(kbd–make–sparse–keymap)

Returns empty keymap

E.g. *(setq new–keymap (kbd–make–sparse–keymap))*
Returns empty keymap

Now that you have a blank keymap, you'll want to start assigning it values.

12–6 modify–keys

```
(defun modify–keys ()
 ; assign keys
(let (new–map)
  ; create a new blank keymap
  (setq new–map (kbd–make–sparse–keymap))
```

```
; assign two new keys to the keymap
(kbd-bind new-map
  "\^Xd" 'date-function      ; bind ^Xd to one function
  "\^[\r" 'new-return-function)    ; bind return key to another function
; tell the doc that it should take the new keymap as well as the normal doc
; one
(tell *document* mid:set-props :keymap (list exp-map kbd-doc-map))
))
```

Suppose you want a character to become a prefix, as ^X is. To make a new character stand as a prefix, you create a new keymap and bind it to a key. For example, suppose you want ^Q to be a prefix key in the *kbd–doc–map*:

12–7 create–prefix–key

```
(defun create-prefix-key ()
(let (prefix-q-map)
  ; create a new keymap
  (setq prefix-q-map (kbd-make-sparse-keymap))
  ; bind it to ^Q
  (kbd-bind kbd-doc-map "\^Q" 'prefix-q-map)
  ; bind ^QA to some function
  (kbd-bind prefix-q-map "A" 'some-function)
))
```

***Function 12–10:* kbd–key–history**

(kbd–key–history)

***Returns*: list of previous keystrokes**

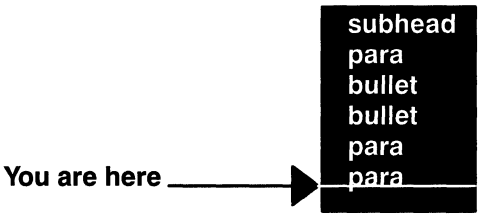
E.g. *(kbd–key–history)*

(#\s #\t #\| #\o #\f #\| #\p #\r #\e #\v #\i #\o #\u #\s #\| #\k #\e #\y #\s #\t #\r #\o #\k #\e #\s #\k #\b #\d #\| ^[#\– #\k #\e #\w #\y #\| ^X # #\B #\a #\c #\k #\S #\p #\a #\c #\e #\| ^X #* #\B #\a #\c #\k #\S #\p #\a #\c #\e #\y #\| ^[#\– #\h #\i #\s #\t #\o #\r #\y #\| ^X #* #\B #\a #\c #\k #\S #\p #\a #\c #\e #\| #\t #\| ^E #\| #\r #\| ^[#\x #\(#\k #\b #\d #\– #\k #\e #\y #\– #\h #\i #\s #\t #\o #\r #\y #\) #\| #\n)*

This function lets you see the keys you have just typed.

The following set of functions allows you to type ^Xx within the text area of a component and have it turn itself into the type of component above it. If you type ^Xx again, the component will turn into the next one above it. So, if you create a *para*, and above it are (in ascending order) *para*, *bullet*, *bullet*, *para*, and *subhead*, the first time you type ^Xx, the component you're in will turn into *bullet*; the second time it will turn into *subhead*, and so on.

```
; set some global variables
(setq *c-list* nil)  ; list of cmpns
(setq *backup-start-cmpn* nil) ; where to start looking for new cmpn
```



The first time you type ^Xx, the para you're in turns to bullet. The second time, it becomes a subhead.

12–8 bkup–2nd–try

```
(defun bkup–2nd–try ()
; were previous keystrokes ^Xx? Get previous keystrokes and compare
(let (klist list–len)
  ; get the history of keystrokes
  (setq klist (kbd–key–history))
  ; get the number of the second to last element on the list of keystrokes
  ; which is the keystroke before this one
  (setq list–len (– (list–length klist) 2))
  ; were the last two characters entered ^X and x?
  (and (equal #\\^X (nth list–len klist))
    (equal #\x (nth (1– list–len) klist)))
))
```

12–9 kbd–doc–cmpn–change

```
(defun kbd–doc–cmpn–change ()
(let (new–cmpn
  (cmpn (doc–point–cmpn)))
  ; Did we just do a ^Xx? If so, we need to go back one more cmpn
  ; If it's our first ^Xx, then push current cmpn on to the list of cmpns
  (if (not (bkup–2nd–try))
    (progn
      (setq *c–list* nil)
      (push (tell cmpn mid:get–name) *c–list*)
      ; capture the current cmpn as the starting point for search upwards
      (setq *backup–start–cmpn* cmpn)))

  ; do the change
  (tell *cmpn–editor* mid:deselect :all)
  (tell *cmpn–editor* mid:select cmpn)
  ; get the previous unused cmpn
```

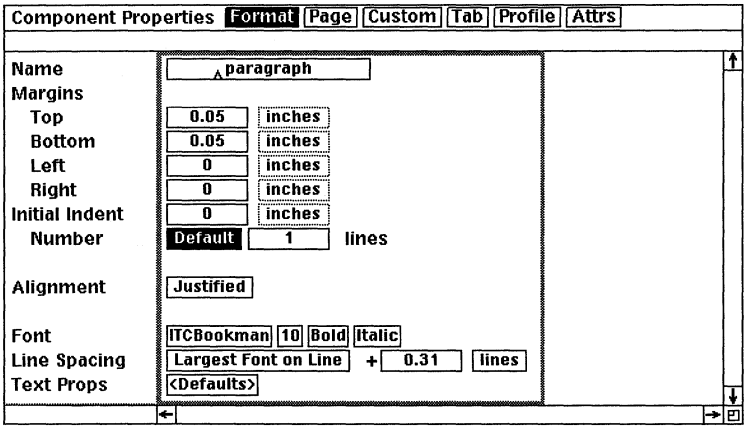
```
  (setq new-cmpn (get-previous-cmpn *backup-start-cmpn*))
  (if new-cmpn
    (progn
      ; capture latest cmpn as the last place we looked so that multiple
      ; ^Xx's will
      ; cycle through cmpns, moving up the doc
      (setq *backup-start-cmpn* new-cmpn)
      (tell *cmpn-editor* mid:change (tell new-cmpn mid:get-name))))
  (doc-flush-queue)
  ))
```

12–10 get-previous-cmpn

```
(defun get-previous-cmpn (c)
; gets previous cmpn
(let (cmpn (thiscmpn (doc-point-cmpn)))
  ; loop until we find a cmpn not already used or until no more left
  (while (and (setq cmpn (tell c mid:get-previous))
          (member (tell cmpn mid:get-name) *c-list* 'string=))
    (if (not cmpn) ; ran out of cmpns
      (progn
        reset list to just this cmpn
        (setq *c-list* (list (tell thiscmpn mid:get-name)))
        (setq cmpn thiscmpn))) ; if at begin of doc, reset to current cmpn
        (setq c cmpn))

  ; record the new cmpn
  (if cmpn
    (push (tell cmpn mid:get-name) *c-list*))
cmpn
 ))
; bind it to ^Xx
(kbd-bind kbd-doc-map "\^Xx" 'kbd-doc-cmpn-change)
```

Chapter 13
Property Sheets

An Interleaf 5 property sheet menu is a dialogue box that contains buttons and input fields. Property sheets provide a convenient and familiar way for users to interact with the system.



A familiar property sheet.

There are, however, limitations to using property sheets from the developer's point of view:

- They cannot contain graphics.
- The types of buttons are already defined.
- Their layout can only follow rigid rules.
- They have to be layed out programmatically, not using the interactive editing tools.

So why use them?

- The pre-existing buttons are powerful.

- They open fast.
- Interleaf users are already familiar with interacting with them.

You might want instead to use an active document as an interface. That will give you more freedom to invent what you want — but it also means that you have to invent it. Once you get the hang of property sheets you'll find them an important part of your kit bag.

Note: The Lisp code for creating property sheets — or "prop sheets" as fans refer to them — is almost certain to undergo drastic change in the new GUI versions of Interleaf 5. It is not yet known how much change prop sheet code will have to undergo to run on the Motif, on Windows and versions of Interleaf 5.

Creating a property sheet

A page of a property sheet is called a *submenu.* A property sheet may have many different submenus which the user accesses by pressing buttons listed on top. Remember, a property sheet menu is the object that has prop sheet submenus, and a submenu is the visible rectangle that users interact with.

Let's take a look at the steps required to create a prop sheet. The first one we create will have no contents. (We'll look at this in logical order, not in the order in which the functions should actually appear in your code; you'll get examples of that later on.)

```
; create a new prop sheet object
(setq blank-prop-sheet (tell-class prop-menu-class mid:new))

; If more than one submenu, specify the text for the button at top
(tell blank-prop-sheet mid:set-props :text "Custom")

; give it a method for opening a submenu
(tell blank-prop-sheet mid:provide mid:open
'blank-submenu-open-method)

; give prop sheet a popup
(tell blank-prop-sheet mid:provide mid:popup 'basic-popup)

; create a submenu
(tell blank-prop-sheet mid:set-props :submenus (ncons
blank-submenu)

; open the prop sheet
(setq blank-prop-sheet-window (tell blank-prop-sheet mid:open
"Blank" 200 100))
```

The last line causes the property sheet to open, with the word "Blank" in its title bar. It will open at a screen position 100 pixels down and 200 pixels from the left. The window itself can be referenced through the variable *blank-prop-sheet-window.*

Actually, it won't really open because we have not yet defined *basic–popup* or *blank–submenu*. Let's do so, by giving them *open* methods.

```
(defun blank-submenu-open-method (submenu)
; tell it to start, with left margin of 1
(tell submenu mid:start 1)
;;
;; here's where we'd put in buttons, text, etc.
;;
; End it
(tell submenu mid:end)
)

(defun basic-popup (menu window)
; Create a popup with a single choice on it: close
  (popup-run
    (popup-create-list
      (popup-create-entry "Close" 'close-prop-sheet nil window)
)))
```

Now, of course, we have to create the close–prop–sheet function:

```
(defun close-prop-sheet (window)
; make the window nil
  (setq blank-prop-sheet-window nil)
  ; do the close
  (tell (car window) mid:close)
)
```

As usual, you should actually put the last three defuns at the beginning before they're referenced by the code that creates the property sheet.

At the end of this chapter, you'll get a simple example and then an extended, working example. But first, let's look at some of the other methods of property sheets.

Function 13–1: new

(tell-class prop-menu-class mid:new)

Returns new prop sheet object

(setq new-prop-sheet (tell-class prop-menu-class mid:new)

Returns prop sheet object

This is the basic method for creating a new property sheet object.

Before the new prop sheet object is created on screen, you must, among other things, open it.

Function 13–2: open

(tell prop–sheet mid:open *title xpos ypos* &optional *width height*

Returns window object if open is successful; else, nil

(tell new–prop–sheet mid:open 150 200 300 400)

Returns window object, and creates prop sheet at location 150 pixels from the left and 200 pixels from the top of the screen. Prop sheet is 300 pixels wide and 400 pixels high.

Usually you will not want to supply the last two arguments when opening a prop sheet so that the system will figure out what the optimal size of the prop sheet is. If you do force the size, then you may size the window so small that some of the buttons are not visible.

The window object that is returned by this method is what you use to send most messages to, including the close message.

Function 13–3: close

(tell window mid:close)

E.g. *(tell my–prop–window mid:close)*
Prop sheet closes

Function 13–4: prop–current–menu

(prop–current–menu)

Returns prop sheet object in current window; if none, then nil

This allows you to get the prop sheet if you have the window.

Function 13–5: prop–current–submenu

(prop–current–submenu)

Returns currently active submenu (or sheet) in the current window

Function 13–6: get–window

(tell prop–sheet–object mid:get–window)

Returns list of all windows currently displayed for prop–sheet–object

E.g. *(tell my–prop–sheet mid:get–window)*
Returns window object

Let's say you open a property sheet object twice, so that two instances of it are displayed onscreen. This method will return a list of all the windows.

Since you frequently will not want to have more than one instance of the prop sheet open, the following will only open the prop sheet if there are no instances already open.

13–1 prop–sheet–already–open

```
(defun prop–sheet–already–open (prop–sheet–obj)
  (if (not (tell prop–sheet–obj mid:get–window))
    (tell prop–sheet–obj mid:open 200 300))
)
```

It's worth pointing out the symptoms of prop sheet code gone wrong. If you make a mistake in the function that draws the prop sheet, it will look like nothing has changed on your desktop, but as you click, you'll find your mouse unresponsive. What has actually happened is that the system has tried to draw a giant propsheet submenu, the size of your desktop, but it failed before it filled in any prop sheet contents. The mouse responds the way it would in any prop sheet without buttons. The solution is to size the prop sheet down as far as it will go and stick it somewhere on your desktop where it won't bother you.

There is a also a set of functions that reports on the state of the prop sheet, allowing the programmer to use some built–in functionality quite easily.

Function 13–7: get–modified–window

(tell prop–sheet–object mid:get–modified *window*)

Returns t if the prop sheet in *window* has been modified.

The primary use of this function is to see whether you should be showing the user a popup that includes "Apply" or "Cancel" as options. If no changes have been made, then "Apply" and "Cancel" don't make sense.

Here is an example of how to make your popup menus sensitive to this information:

13–2 has–propsheet–changed

```
; create popup method
(defun my–popups (prop–sheet window)
(let (changes–popup no–changes–popup)
  ; create popup for when there have been changes
  (setq changes–popup
    (popup–create–list
      (popup–create–entry "Apply" 'my–apply nil window)
      (popup–create–entry "Cancel" 'my–cancel nil
        window)))
  ; create popup if no changes
  (setq no–changes–popup
    (popup–create–list
      (popup–create–entry "Close" 'my–close nil window)))
      ; if it's been modified
  (if (tell prop–sheet mid:get–modified window)
    (popup–run changes–popup)
     ; else
    ( no–changes–popup))
))

; create the prop sheet object
(setq my–prop–sheet (tell–class prop–menu–class mid:new))
```

```
; give the prop sheet a popup method
(tell my-prop-sheet mid:provide mid:popup 'my-popups)
```

Of course, you would now need to provide the functions your popups are calling.

You could use the same basic technique to give each submenu (sheet) special popups.

Submenu contents

Now that you know how to create a property menu with a submenu, you need to know how to give the submenu some contents.

The contents of a submenu are specified in the function that provides the submenu's open method, as defined when the submenu was first given to the prop sheet.

Submenus have a left margin that is usually used to list the topic of the line. You declare the size of that margin when you tell the submenu to start:

Function 13–8: start

(tell *submenu* mid:start *margin*)

Tells submenu to start and creates left margin of *margin*

(tell submenu1 mid:start 12)

Between the start and end commands, you define a submenu by describing each line, in order.

You create a new line on a submenu with the following method:

Function 13–9: line

(tell *submenu* mid:line *text*)

Creates new line on submenu and puts *text* into left margin

E.g. *(tell submenu1 mid:line "Font")*
Creates line with "Font" in left margin

To create a line with no text in the margin, use *(tell submenu mid:line "")*.

You can show or hide a line:

Function 13–10: is–effective

(tell *submenu* mid:is–effective *line–number text*)

Creates a line on submenu at line–number and puts in text; if nil, hides the line at line–number

E.g. *(tell submenu1 mid:is–effective 5 "Super Secret")*
Creates a fifth line with the text "Super Secret"

E.g. *(tell submenu 1 mid:is–effective 5 nil)*
Hides fifth line

In order for this change to take effect, the property sheet has to be told to redraw:

Function 13–11: redraw

(tell *submenu* mid:redraw *submenu1*)

Redraws *submenu* with *submenu1*'s contents

E.g. *(tell (prop–current–menu) mid:redraw (prop–current–menu))*
Redraws current submenu

You can put text anywhere in the submenu through the *text* method:

Function 13–12: text

(tell *submenu* mid:text *text X &optional* :border)

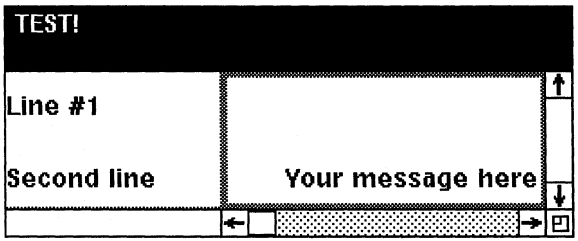
Puts *text* into *submenu* at position *X*, with *border* if keyword is used

E.g. *(tell submenu1 mid:text "Press Return" 10 :border t)*
Puts "Press Return" into submenu1, with a border, at column 10 of the current row

The current row is the one your submenu open method is currently describing. For example:

13–3 open–my–submenu

```
(defun open–my–submenu (submenu)
  ; start the submenu, and give it a left margin of 15
  (tell submenu mid:start 15)
  ; create a line with the word "Line #1"
  (tell submenu mid:line "Line #1")
  ; create a blank line
  (tell submenu mid:line "")
  ; create a second line
  (tell submenu mid:line "Second line")
  ; Add some text inside the prop sheet, 4 spaces over
  (tell submenu mid:text "Your message here" 4)
  ; end it
  (tell submenu mid:end)
)
```



Really boring sample prop sheet.

Remember, of course, that to get this function to work, you'd have to provide the necessary housekeeping described at the beginning of this chapter: create a prop sheet menu, give it

some popups, give it at least one submenu, give the submenu an open method (which is what the *open–my–submenu* example above does), and open the prop sheet. Here is code that will do the job

13–4 sample–simple–propsheet

```
; create new prop sheet submenu
(setq mysubmenu (tell–class prop–submenu–class mid:new))

; give it methods for opening the two submenus
(tell mysubmenu mid:provide mid:open 'open–my–submenu)

(setq prop–sheet–menu (tell–class prop–menu–class mid:new))

; give the prop sheet object the two submenus
(tell prop–sheet–menu mid:set–props :submenus (ncons mysubmenu))


; open the prop sheet
(setq test–menu (tell prop–sheet–menu mid:open "TEST!" 400 400))
```

If you don't want to have to figure out what the left margin ought to be, you can create a list of all the words you want in the left margin, and have the system calculate the longest one:

13–5 get–longest–word

```
(defun get–longest–word (word–list)
(let (item (longest 0))
  (while (setq item (pop word–list))
    (if (> (string–length item) longest)
      (setq longest (string–length item))))
 longest
))
```

Then you would alter the open–my–submenu function as follows:

13–6 open–my–submenu2

```
(defun open–my–submenu (submenu)
(let (line–words)
  ; create list of words
  (setq line–words (list "Line #1" "Second line"))
  ; start the submenu, and calulate longest line
  (tell submenu mid:start (get–longest–word line–words))
  ; fill in the rest ...
  ; ...
  (tell submenu mid:end)
))
```

Buttons

There are three basic types of buttons available on a prop submenu. All three invoke a specified function when pressed. A toggle button simply turns on or off when selected. A list button is a set of radio buttons; turn one on and all the others turn themselves off. A text button displays text in it; with each click, the text changes to the next string in the list, cycling indefinitely.

Let's look at each of the three.

Toggle buttons

Function 13–13: toggle button

(tell *submenu* mid:button *string position width function* :type :toggle *&optional* :initial *state* :arg *arguments*)

Creates toggle button at position *position* of *width* characters wide, with text content of *string*. It invokes *function* when pressed. It has initial *state* of on or off (t or nil) and passes to *function* additional *arguments*.

E.g. *(tell submenu1 mid:button "Italics" 5 0 'italic–fcn :type :toggle :initial nil :arg my–count)*
Creates button with word "Italics" in it at column 5. Because width is 0, it calculates the right width. It is initially in off state. If selected, it invokes italic–fcn and passes it the argument my–count.

Notice that if the width is 0, it calculates the appropriate width.

With a toggle button, the function that gets invoked gets passed the old state and the new state, as t or nil. If you pass it some additional arguments, they of course show up. A typical function for a toggle button might look like this:

13–7 italic–fcn

```
(defun italic–fcn (old new &optional times)
  (if (< times 10)
    (progn
      ; set italic font on or off, appropriately
      (tell *text–editor* mid:set–props :italic new))
    ; else
      (stk–open "Style Violation! You've changed italics
       more than 10 times."))
t)
```

Note. Button functions usually ought to return t. Failure to do so may result in unexpected results. The function is passed the current setting (t or nil) and what the setting will be if the function returns non-nil. So, if you forget to give your function something to return, it will return something perhaps unintended, and will not toggle itself if the return happens to be nil.

List buttons

(tell *submenu* mid:button *string position width function* :type :list *list* *&optional* :initial number :arg *arguments*)

Creates list of buttons *list* starting at position *position* of *width* characters wide. It invokes *function* when button number *number* is chosen. It passes to *function* additional *arguments.*

E.g. *(tell submenu1 mid:button "Swiss" "Thames" "Courier" 5 0 'family–fcn :type :list :initial 0 :arg permission)*
Creates list of buttons "Swiss" "Thames" and "Courier," with the Swiss button on. It begins at column 5. Because width is 0, it calculates the right width. If selected, it invokes family–fcn and passes it the argument permission.

The function that gets called gets passed the number of the button that was clicked (zero-based numbering). Here is a sample function:

13–8 family–fcn

```
(defun family–fcn (choice)
; sets font to chosen family
(let (family)
  (cond
    ((= 0 choice) (setq family "Swiss"))
    ((= 1 choice) (setq family "Thames"))
    ((= 2 choice) (setq family "Courier"))
  )
  ; set family
  (tell *text–editor* mid:set–props :family family)
  ; return t
t
))
```

Another way to handle this would be to set up the button as follows:

```
; create  a list of button names
(setq buttons (list "Swiss" "Thames" "Courier"))
; use this list to build the buttons
(tell submenu1 mid:button :buttons buttons 5 0 'family–fcn :type :list :initial
0 :arg buttons)
```

This passes the button list to the function where you can decode it as follows:

```
(defun family–fcn (choice buttons)
  ; use the number of the button to find the choice on the list
  (tell *text–editor* mid:set–props :family (nth choice
    buttons))
```

t
)

This uses the *nth* function to retrieve the right button from the list of buttons.

Text buttons

Text buttons allow the user to cycle through a set of choices. They provide a convenient interface when there are too many choices to show in a series of list buttons. In addition, a text button can have a popup menu that lists all available choices, so the user can go straight to the right one by holding down the menu button while on the text button (although this takes additional work on the programmer's part).

The basic way of creating a text button is straightforward:

Function 13–14: text button

(tell *submenu* mid:button *text position width function* :type :text *&optional* :initial *state* :arg *argument*)

Creates text button at *position* with *text*, *width* wide, invoking *function*. Passes *function argument*. Button reflects *state* (t or nil).

E.g. *(setq family–list (fn–families))*
*Returns list of font families available in *document**

(tell submenu1 mid:button "Swiss" 5 0 'families–fcn :type :text :arg family–list)
Creates text button five characters in, automatically assuming the right size. Invokes families–fcn when selected, passing it the mouse button pressed on the button; also passes family–list.

If you click on a text button, the function that is invoked will receive as an argument a symbol indicating which mouse button you pushed: *:select*, *:menu* or *:extend*. In general, the way Interleaf 5's own interface works, the select button shows you the next item on a list, the extend button shows you the previous, and the menu button shows you the list in popup form. In the next few pages, we'll reproduce that sort of interface, in this case using a list of available fonts as our example. (In the extended example at the end, you'll see how to get the list of fonts.)

As with the other types of buttons, specifying a width of zero causes the button to size itself to fit the text, but since that text may alter as the user cycles through the button choices, the size of the button will alter. Therefore, it may make sense to specify a width so that the button will always be the same size.

Here is what the function that gets invoked might look like:

13–9 families–fcn
(defun families–fcn (old mouse–button families)
(let (return)
; old contains original button text

```
  ; act differently depending on which button was pressed
  (cond
    ((eql mouse-button :select) (setq return (get-next old  families)))
    ((eql mouse-button :menu) (setq return (popup-run
      *families-popup*)))
    ((eql mouse-button :extend) (setq return (get-prev old families))))
return))
```

Let's look at this more closely.

The *cond* says that if the select button has been pressed, we should set *return* to whatever the function *get-next* returns. Let's create a *get-next* function:

13-10 get-next

```
(defun get-next (current fam-list)
(let (pos next-string)
  ; get the position of current choice in family list
  (setq pos (position current fam-list :test 'string=))
  ; if not the last one, then get the next
  (if (not (eql (1+ pos) (list-length fam-list)))
    (setq next-string (nth (1+ pos) fam-list))
    ; else, it's the last one, so get the first
    (setq next-string (car fam-list)))
; return string
next-string
))
```

This will make the return equal to the value of the next string we want; whatever it returns is what's put into the text button. So, now we can use the select button to cycle through the choices.

The *families-fcn* should also let us move backwards through the list. This requires writing a function very similar to *get-next*:

13-11 get-prev

```
(defun get-prev (current fam-list)
(let (pos prev-string)
  ; get the position of current choice in family list
  (setq pos (position current fam-list  :test 'string=))
  ; if not the first one, then get the previous
  (if (not (eql pos 0))
    (setq prev-string (nth (1- pos) fam-list))
    ; else, get last
    (setq prev-string (last fam-list)))
; return string
```

```
  prev-string
  ))
```

Finally, we have to write a function that will create a popup object named *families-popup* so that the menu button can run it. This function needs to be called before the user gets to press the menu button while over the button we've created.

13–12 make-font-popup

```
(defun make-font-popup-obj (font-list)
; Make popup object out of list of fonts
(let (f-list item (p-list nil))
  ; create copy of font list
  (setq f-list (copy-list font-list))
  ; go through list, making list of entries
  (while (setq item (pop f-list))
    (push  (popup-create-entry item item) p-list))
  ; make the popup object by applying the list through
  ; popup-create
  (setq *families-popup* (apply #'popup-create-list
    p-list))
))
```

In the example we've been building, the select button cycles down a list, the extend button cycles up the list, and the menu button shows you the list in popup form. That is how Interleaf 5's own prop sheet interface works. Your application need not follow these conventions. But if you choose to violate the conventions, you probably should have a good reason for doing so.

Fields

Prop sheets can contain input fields into which the user can type information.

Function 13–15: prop-sheet field

(tell submenu mid:field *text position width function* &*optional* :arg *value* :type {:text :popup :integer} :maximum *number*)

Create field with *text* as its content at *position*, *width* characters wide, that invokes *function*, passing *value* as an argument. The field will accept *number* of characters if specified.

E.g. *(tell submenu1 mid:field "Swiss" 10 20 'check-font :arg 4 :type :text :maximum 30)*
Creates field in submenu1 at position 10. It's 20 characters wide and starts with the text "Swiss" showing. It calls check-font and passes it the value 4. It is of type :text and will accept a total of 30 characters as input.

Users get the standard Interleaf user interface text editing capabilities in a field; for example, the arrow keys work to move within the typed-in text and edit it. When the user is done

entering text (hits a return or moves the mouse out of the field), the system invokes the named function, passing it the old contents as well as the new. Most likely, you'll write the function that checks to make sure that the entry makes sense (as defined by you, the developer), and will take corrective action if it does not.

The system already knows how to do some type checking. If you specify the *:type* as *:integer*, and a user enters something other than an integer, the system will put a message in the message bar of the prop sheet window and will return the text in the field to its previous state. Types *:text* and *:popup* accept any input and treat it as a string.

The *:maximum* value ("number" in the example) tells it how long the string can be. This can be longer than the width of the field; the system will automatically scroll.

Popup–type fields operate a little differently. What is passed to the function is the current text in the field and the value *:popup,* which is t if the user has pressed the menu button and nil otherwise. This allows the user to get a popup menu of choices which, if selected, will automatically fill in the field.

For example, consider the previous example which starts with "Swiss" as its contents. The function it calls might look like this:

13–13 field–fcn

```
(defun field-fcn (old new)
  ; User pressed menu button?
  (if (eql new :popup)
    (progn
      (popup-run
        (popup-create-list
          (popup-create-entry "Swiss" "Swiss")
          (popup-create-entry "Thames" "Thames")
          (popup-create-entry "Courier" "Courier"))))
    ; else, not a popup, then check if it's on list
    (if (not (member new (list "Swiss" "Thames" "Courier") 'equal))
        (stk-open "Not an acceptable font choice. Try again.")))
)
```

Multiple submenus

If there is more than one submenu, they will show up as buttons at the top of the prop sheet. To create multiple submenus, you have to create the submenu object, give it an open method, state what text you want used in the button at the top, and give the property sheet a list of all the submenus. The following code should get you started.

```
; create two new prop sheet objects
(setq submenu1 (tell-class prop-submenu-class mid:new))
```

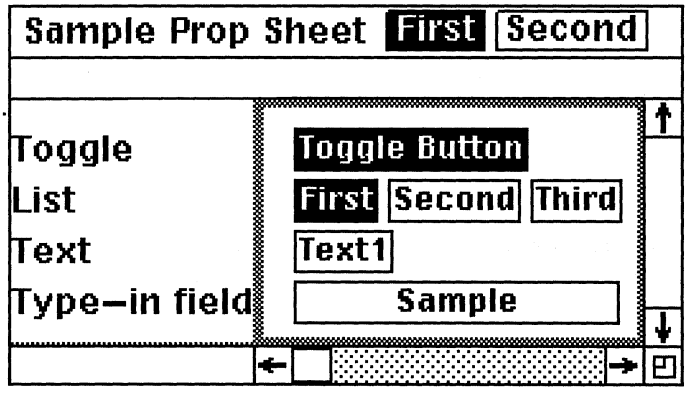
```
(setq submenu2 (tell-class prop-submenu-class mid:new))

; Create two submenu buttons for the top of the prop sheet
(tell submenu1 mid:set-props :text "First Submenu")
(tell submenu2 mid:set-props :text "Second Submenu")

; give it methods for opening the two submenus
(tell submenu1 mid:provide mid:open 'submenu1-open)
(tell submenu2 mid:provide mid:open 'submenu2-open)
; (remember to provide these open methods eval'ing this code!)

(setq prop-sheet-menu (tell-class prop-menu-class mid:new))

; give the prop sheet object the two submenus
(tell prop-sheet-menu mid:set-props :submenus (list
  submenu1 submenu2))

; open the prop sheet
(setq test-menu (tell prop-sheet-menu mid:open "TEST!" 400 400))
```

Sample blank prop sheet with two submenus.

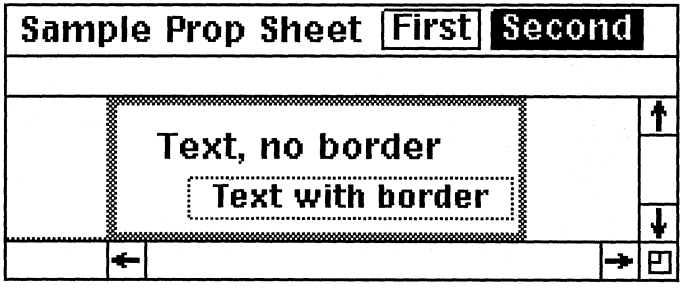
Sample simple propsheet

Now let's look at a sample property sheet that uses all the different buttons unconstrained by the need to do anything useful.

Here is the property sheet that we will be constructing:



First sheet (submenu) of sample property sheet.



Second sheet (submenu) of sample property sheet.

We'll begin by defining the functions that will get called as the user interacts with the property sheet.

13–14 toggle-function

```
(defun toggle-function (old new &optional data)
  (stk-open (concat "Turning toggle " (if new "on" "off")))
t)
```

13–15 list-function

```
(defun list-function (old new which)
  (stk-open (concat "You chose button #" (itoa new) " (zero based).")))
t)
```

13–16 text-function

```
(defun text-function (old mouse choices)
  (let (mouse-text number next)
    ; translate mouse button symbol to string
    (cond
```

```
      ((eql mouse :select) (setq mouse-text "select"))
      ((eql mouse :popup) (setq mouse-text "menu"))
      ((eql mouse :extend) (setq mouse-text "extend")))
    (stk-open (concat "Old text was " old " and you pressed " mouse-text
    " button."))
  ; get next on list, or first if at end
  (setq number (position old choices :test 'string=))
  (if (= (1+ number) (list-length choices))
    (setq number 0)
    ; else, increase number by one
    (inc number))
  ; get the right one from the list of choices
  (setq next (nth number choices))
next))
```

13–17 type-in-function

```
(defun type-in-function (old new)
  (stk-open (concat "You typed in: " new))
)
```

Now we'll define what the prop sheets will look like and where each button will go.

13–18 submenu1-open

```
(defun submenu1-open (sheet)
  ; start with label margin of 12
  (tell sheet mid:start 12)

  ; create toggle button, initially on
  (tell sheet mid:line "Toggle")
  (tell sheet mid:button "Toggle Button" 2 0 'toggle-function :initial t)

  ; create list of buttons, with third initially on
  (tell sheet mid:line "List")
  (tell sheet mid:button (list "First" "Second" "Third") 2 0 'list-function :type
    :list :initial 2 :arg t)

  ; create text button with "Text1" chosen as default
  (tell sheet mid:line "Text")
  (tell sheet mid:button "Text1" 2 0 'text-function :type :text :arg (list "Text1"
  "Text2" "Text3"))

  ; create text type-in field with word "Sample" in it
  (tell sheet mid:line "Type-in field")
  (tell sheet mid:field "Sample" 2 15 'type-in-function :type :text)
```

```
  ; end it
  (tell sheet mid:end)

)
```

13–19 submenu2–open

```
(defun submenu2–open (sheet)
(tell sheet mid:start 5)
 ; on second submenu, only place some text; blank line
(tell sheet mid:line "")
(tell sheet mid:text "Text, no border" 2)
; blank line
(tell sheet mid:line "")
; place some text with a border
(tell sheet mid:text "Text with border" 4 :border t)

  ; end it
  (tell sheet mid:end)

)
```

Now we'll define what happens when you make a choice from the property sheet's popup menu, and then we'll create the popup menu object itself.

13–20 test–apply

```
(defun test–apply (&optional data)
  (stk–open "Here's where you would actually do an apply"))
```

13–21 test–close

```
(defun test–close (window)
  (tell (car window) mid:close))
```

13–22 sample–popup

```
(defun sample–popup (menu window)
; create a popup with an apply or close button
  (popup–run
    (popup–create–list
      (popup–create–entry "Apply" 'test–apply nil window)
      (popup–create–entry "Close" 'test–close nil window)))
)
```

Now we'll do the work that defines the property sheet and opens it.

13–23 make sample prop sheet

```
; make two new prop sheet submenu objects
```

```
(setq submenu1 (tell-class prop-submenu-class mid:new))
(setq submenu2 (tell-class prop-submenu-class mid:new))

; create two submenu buttons for the top of the prop sheet
(tell submenu1 mid:set-props :text "First")
(tell submenu2 mid:set-props :text "Second")

; give each submenu a method for opening
(tell submenu1 mid:provide mid:open 'submenu1-open)
(tell submenu2 mid:provide mid:open 'submenu2-open)

; create new prop sheet
(setq new-prop-sheet (tell-class prop-menu-class mid:new))

; give the prop sheet the two submenus
(tell new-prop-sheet mid:set-props :submenus (list submenu1
submenu2))

; give prop sheet a popup (defined above)
(tell new-prop-sheet mid:provide mid:popup 'sample-popup)

; open the prop sheet
(setq test-menu (tell new-prop-sheet mid:open "Sample Prop Sheet"
300 200))
```

Sample propsheet application

Now let's write an actual application using a property sheet interface. In order to use as many of the different types of buttons as possible, this example perhaps goes a little overboard. But it is actually a useful function: it allows the user to apply new font properties to all of the objects in a document in one fell swoop. So, if you decide you'd like to change all your fonts to Swiss, while maintaining the various point sizes, bolding, etc., this prop sheet will let you.

The code that actually opens the prop sheet relies upon other functions having been loaded first. So, leave that code for the end of the program.

Let's begin by looking at the open methods for two submenus. These describe what the property sheets will look like.

For reference purposes, I have numbered the buttons and the corresponding functions they invoke.

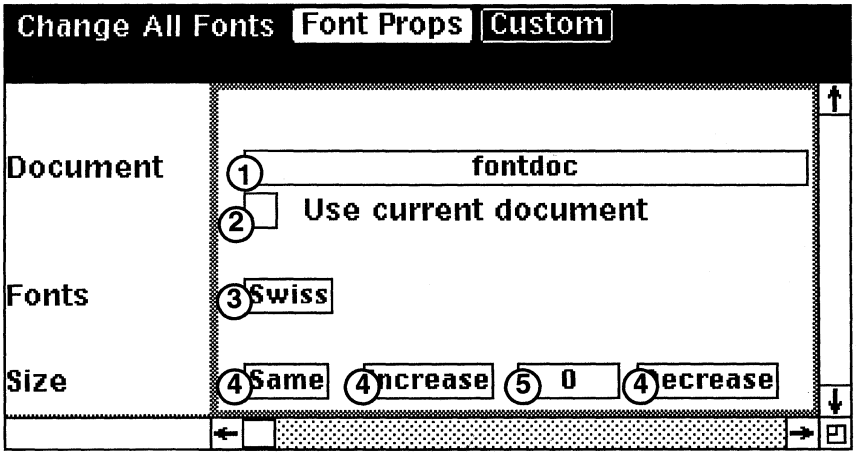
```
;;-------------------------------------;;
Design the first submenu
```

13–24 change–fonts–submenu–open

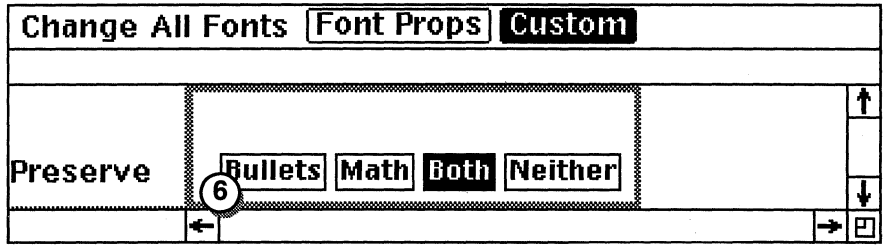
```
(defun change-fonts-submenu-open (sheet)
  ; start
  (tell sheet mid:start 12)
  ; insert blank line
  (tell sheet mid:line "")

  ; 1.Create text type-in field with current doc name in it
  (tell sheet mid:line "Document")
  (tell sheet mid:field (get-current-doc-name) 2 32
    'doc-name-fcn :type :text)

  ; 2.Create toggle button to auto-insert current doc name
  (tell sheet mid:line "")
  (tell sheet mid:button "" 2 2 'insert-current-name-fcn
    :type :toggle :initial *cur-name* )
  ; add explanatory text
  (tell sheet mid:text "Use current document" 5)

  ;3. Create text button that cycles through fonts, or gives popup
  (tell sheet mid:line "")
  (tell sheet mid:line "Fonts")
  (tell sheet mid:button *family* 2 0 'family-popup-fcn :type :text)

  ; 4.Buttons for increase/decrease font size, or stay the same
  (tell sheet mid:line "")
  (tell sheet mid:line "Size")
  (tell sheet mid:button "Same" 2 0 'same-size-fcn :type
    :toggle :initial *same-size*)
  (tell sheet mid:button "Increase" 10 0 'inc-size-fcn
    :type :toggle :initial *inc-size*)
  (tell sheet mid:button "Decrease" 25 0 'dec-size-fcn
    :type :toggle :initial *dec-size*)

  ; 5.Type-in field displays font increment
  (tell sheet mid:field (itoa *font-boost*) 18 5
    'font-boost-fcn :type :text )

  ; end
  (tell sheet mid:end)
)
```

```
;;-----------------------------
;; Design the second submenu

13-25 change-fonts-custom-submenu-open
(defun change-fonts-custom-submenu-open (sheet)
  ; start
  (tell sheet mid:start 10)

  ; 6.Radio buttons to save bullets, math, both or neither
  (tell sheet mid:line "")
  (tell sheet mid:line "Preserve")
  (tell sheet mid:button (list "Bullets" "Math" "Both"
    "Neither") 2 0 'save-bullets-math-fcn :type :list
    :initial *save-button*)
  ; end
  (tell sheet mid:end)
)
```

The following screen captures are numbered according to the code that created them:



Submenu 1.



Submenu 2.

Now let's look at the functions the prop sheet invokes as the user interacts with it.

```
; 1 ----------------------------
; Checks typed-in name to see if it matches an open doc
;
```

13-26 doc-name-fcn

```
(defun doc-name-fcn (old new)
; Checks typed in name to see if it's an open doc
; This gets called when you finish typing in a doc name
(let (win-list win obj doc name (obj-list nil))
  ; get list of all windows
  (setq win-list (tell *wn-wmgr* mid:get-props :windows))
  ; look through windows
  (while (setq win (pop win-list))
    (if (and (setq obj (tell win mid:get-object))
      (is-of-class obj doc-editor-class))
      (progn
        (setq doc (tell obj mid:get-document))
        (setq name (tell doc mid:get-name))
        (push name obj-list))))
  ; if name not on list, then put notice in msg bar
  (if (not (member new obj-list 'string=))
    (wn-message nil (concat "No document named " new
      "."))))
new
))

; 2 ----------------------------
; Gets current doc's name and fills in the blank with it
;
```

13-27 get-current-doc-name

```
(defun get-current-doc-name ()
```

```
; Gets name of current doc to which changes will be
; applied
; Called when propsheet opens to fill in the blank
(let ((name "No document open"))
  ; if a doc is open, get its name
  (if (doc-current-icon)
      (setq name (tell *document* mid:get-name)))
name
))
```

13–28 insert-current-name-fcn

```
(defun insert-current-name-fcn (old new)
; Redraws current menu, at which point it inserts name of current doc
; blink it
  (setq *cur-name* t)
  (tell (prop-current-menu) mid:redraw (prop-current-window))
  (setq *cur-name* nil)
  (tell (prop-current-menu) mid:redraw (prop-current-window))

t)

; 3 ----------------------------
; Builds font list, lets user cycle through names
;
```

13–29 get-font-list

```
(defun get-font-list ()
: Returns list of font names. Ignores system fonts
(let (fn-list f item (font-list nil))
  ; skips car of list returned by fn-families because
  ;  that's where the system fonts are listed
  (setq fn-list (cdr (fn-families)))
  ; cycle through the list, flattening it so there are no
  ; nested lists
  (while (setq item (pop fn-list))
     (while (setq f (pop item))
       (push f font-list)))
 font-list
))
```

13–30 family-popup-fcn

```
(defun family-popup-fcn (old new)
; Cycle up or down, or show popup of all families
; old contains original button text
(let (family return)
```

```
  ; act differently depending on which button was pressed
  (cond
    ((eql new :select) (setq return (get-next old
      *font-list*)))
    ((eql new :popup) (setq return (popup-run
      *families-popup*)))
    ((eql new :extend) (setq return (get-prev old
      *font-list*))))
(setq *family* return)
return ))
```

13–31 get–next

```
(defun get-next (current fam-list)
;gets next entry on list of font families
(let (pos next-string)
  ; get the position of current choice in family list
  (setq pos (position current fam-list :test 'equal))
    ; if not the last one, then get the next
    (if (not (eql pos (1- (list-length fam-list))))
      (setq next-string (nth (1+ pos) fam-list))
    ; else, get first
    (setq next-string (car fam-list)))
; return string
next-string
))
```

13–32 get–prev

```
(defun get-prev (current fam-list)
 ; gets previous entry on list
(let (pos prev-string)
  ; get the position of current choice in family list
  (setq pos (position current fam-list :test 'equal))
  ; if not the last one, then get the next
  (if (not (eql pos 0))
    (setq prev-string (nth (1- pos) fam-list))
    ; else, get last
    (progn
      (setq prev-string (nth (1- (list-length fam-list)) fam-list)))
  )
; return string
prev-string
))
```

```
; ---------------------------- 4
; Increase/decrease the increment
```

13–33 same–size–fcn

```
(defun same–size–fcn (old new)
  (setq *font–boost* 0)      ; zero the value
  ; flash the button by redrawing window twice
  (setq *same–size* t)       ; set button value
  (tell (prop–current–menu) mid:redraw
    (prop–current–window))
  (setq *same–size* nil)
  (tell (prop–current–menu) mid:redraw
    (prop–current–window))
t)
```

13–34 inc–size–fcn

```
(defun inc–size–fcn (old new)
  (inc *font–boost*)         ; boost the value
  ; flash the button by redrawing window twice
  (setq *inc–size* t)        ; set button value
  (tell (prop–current–menu) mid:redraw
    (prop–current–window))
  (setq *inc–size* nil)
  (tell (prop–current–menu) mid:redraw
    (prop–current–window))
t)
```

13–35 dec–size–fcn

```
(defun dec–size–fcn (old new)
  (dec *font–boost*)         ; decrease the value
  ; flash the button by redrawing window twice
  (setq *dec–size* t)        ; set button value
  (tell (prop–current–menu) mid:redraw
    (prop–current–window))
  (setq *dec–size* nil)
  (tell (prop–current–menu) mid:redraw
    (prop–current–window))
t)
```

```
; ---------------------------- 5
; Accept typed–in number as the amount to change the size
```

13–36 font–boost–fcn

```
(defun font–boost–fcn (old new)
```

```
; boost the global containing amount to increase/decrease ; font size
  (setq *font-boost* (atoi new))
t
)

; ---------------------------- 6
; Radio buttons preserve bullets, math, both or neither
13-37 save-bullets-math-fcn
(defun save-bullets-math-fcn (old new)
; sets variables used when changes are applied
    (cond
    ((eql new 0) (progn
          (setq *save-bullets* t)
          (setq *save-math* nil)))
    ((eql new 1) (progn
          (setq *save-bullets* nil)
          (setq *save-math t)))
    ((eql new 2) (progn
          (setq *save-bullets* t)
          (setq *save-math* t)))
    ((eql new 3) (progn
          (setq *save-bullets* nil)
          (setq *save-math* nil))))
; save it in global and return it
  (setq *save-button* new)
)

;================================
;                  POPUP FUNCTIONS
; Apply and Close functions called by prop sheet's popups


;------------------------- APPLY

13-38 is-bullet
(defun is-bullet (f)
; is cmpn's family a bullet? Dumb way of finding out:
; Checks if name is like "Symbols" or "Dingbats"
(or (string-contained "Symb" (tell f mid:get-props
 :family))
  (string-contained "Ding" (tell f mid:get-props
    :family)))
)
```

13–39 is–math

```
defun is–math (f)
; is cmpn's family a bullet?
(string–contained "Math" (tell f mid:get–props :family))
)
```

13–40 propsheet–apply

```
(defun propsheet–apply (&optional data)
; Big apply function. Builds new font and applies to all.
  (let ((name–list nil) df newfont family)
    ; post a warning
    (if (not (stk–open (format nil "Are you sure you want to change all fonts
in\n~D\nto\n~A?\nNOT recoverable!" (tell (doc–current–icon)
mid:get–name) *family*) :yes–no))
      (quit))
    (setq pool (name–find–pool doc–cmpn–class *document*))
    ; get list of name heads
    (setq child (tell pool mid:get–child :along :name))
    (while child
      (push child name–list)
      (setq child (tell child mid:get–next))
    )
    ; go through list of name heads and find all instances
    (while (setq m (pop name–list))
      ; go through all instances
      (setq c (tell m mid:get–child))
      (while c
        ; create appropriate doc font
        (setq df (tell c mid:get–props :doc–font))
        ; if preserve bullets or math, don't change family
        (if (or
          (and *save–math* (is–math df))
          (and *save–bullets* (is–bullet df)))
            (setq family (tell df mid:get–props :family))
          ; else, make family into new family
          (setq family *family*))
        ; set size and check for range of 2–200
        (setq size (+ *font–boost* (tell df mid:get–props
          :size)))
        (if (< size 1)
          (setq size 2))
        (if (> size 200)
          (setq size 200))
```

```
        ; make new font
        (setq newfont (tell-class doc-font-class mid:new
          :doc-font df :family family :size size)
        ; apply new font
        (tell c mid:set-props :doc-font newfont)
        (setq c (tell c mid:get-next :along :name))))
      ; reset size to 0
      (setq *font-boost* 0)
      (setq *boost-button* 0)
      ; reset button to "same" by redrawing the window
      (tell (prop-current-menu) mid:redraw (prop-current-window))
))

; CLOSE----------
; close method accessed through popup
13-41 close-prop-sheet
(defun close-prop-sheet (window)
  ; make the window nil
  (setq change-fonts-window nil)
  ; do the close
  (tell (car window) mid:close)
)

13-42 make-font-popup-obj
(defun make-font-popup-obj (font-list)
; Make popup object out of list of fonts
(let (f-list item (p-list nil))
  ; create copy of font list
  (setq f-list (copy-list font-list))
  (while (setq item (pop f-list))
    (push  (popup-create-entry item item) p-list))
  (setq *families-popup* (apply #'popup-create-list
    p-list))
))

;;; Create popups for prop sheet window
13-43 change-fonts-popup
(defun change-fonts-popup (menu window)
; Create a popup with a single choice on it: close
  (popup-run
    (popup-create-list
      (popup-create-entry "Apply" 'propsheet-apply nil
        window)
```

```
    (popup-create-entry "Close" 'close-prop-sheet nil window)
)))

;;;; Make some globals
; list of all available fonts
(setq *font-list* (get-font-list))
; amount to raise/lower font
(setq *font-boost* 0)
; font size boost button amt
(setq *boost-button* 0)
; popup object lists all fonts
(setq *families-popup*
  (make-font-popup-obj *font-list*))
; make current-doc-name button off
(setq *cur-name* nil)
; make family button show "Swiss"
(setq *family* "Swiss")
(setq *same-size* nil)          ; make same-size button off
(setq *inc-size* nil)           ; make inc-size button off
(setq *dec-size* nil)           ; make dec-size button off
(setq *save-button* 2)          ; make both bullets and math button on
(setq *save-bullets* t)         ; save bullets
(setq *save-math* t)            ; save math

;-------------------------------------;
Create the prop sheet

; create a new prop sheet object
(setq change-fonts-submenu (tell-class prop-submenu-class
  mid:new))
(setq change-fonts-custom-submenu (tell-class
  prop-submenu-class mid:new))

; create two submenu buttons for the top of the prop sheet
(tell change-fonts-submenu mid:set-props :text "Font
  Props")
(tell change-fonts-custom-submenu mid:set-props :text
  "Custom")

; give it methods for opening the two submenus
(tell change-fonts-submenu mid:provide mid:open
  'change-fonts-submenu-open)
(tell change-fonts-custom-submenu mid:provide mid:open
  'change-fonts-custom-submenu-open)
```

```
; create prop sheet obj
(setq change-fonts-menu (tell-class prop-menu-class
  mid:new))

; give prop sheet a popup
(tell change-fonts-menu mid:provide mid:popup
  'change-fonts-popup)

; give the prop sheet the two submenus
(tell change-fonts-menu mid:set-props :submenus (list
  change-fonts-submenu change-fonts-custom-submenu))

; open the prop sheet
(setq change-fonts-window (tell change-fonts-menu mid:open "Change
All Fonts" 600 400))
```

Yes, that was a mega-example. But it contains many of the different sorts of buttons you can use. Go over it slowly, have a nice warm bath, and you'll be refreshed and ready to try out your own prop sheets.

(Don't forget: If you make an error while you are coding a prop sheet, there's a good chance you'll be left with a semi-visible prop sheet occupying your entire screen. Size it as small as it will go and pretend it's not there.)

Chapter 14
Tables

Access to tables is similar to access to other objects in a document. There is even an object known as the **table–editor** which helps make tables much easier to work with. The only real obstacle is understanding the rather complex structure of Interleaf tables.

Table functions

A table cell is really a graphics frame. Cells appear to be text elements because, by default, when the user selects a cell, she is immediately dropped into a microdocument. If the user, however, selects the cell and changes it from object to diagram the user can now operate in that frame as in an ordinary frame. But no matter what the cell is set to, internally it is a frame.

To get a particular table, you can get the first table in a document and then navigate to the rest.

Function 14–1: get–first
(tell–class doc–table–class mid:get–first)
Returns: first table in document

Now that you have a table, you can get all the rest of them by using *get–next*.

14–1 visit–every–table

```
(defun visit–every–table ()
(let (table)
  ; get first table
  (setq table (tell–class doc–table–class mid:get–first))
  ; visit the rest of them
  (while table
  ; do something here with the table
    (do–something table) ; made–up function
    ; get next table
    (setq table
```

```
    (tell table mid:get-next))
   )
  ))
```

You could also work backwards by using *(tell-class doc-table-class mid:get-last)* and *(tell table mid:get-previous)*.

Now that you have a table, you can get its props.

Function 14-2: get-props
(tell *table* mid:get-props &*optional* prop)
Returns properties

This works the same way all the other *get-props* do.

Here are the properties you can get or set:

Property	Type
:number-of-rows	integer
:number-of-columns	integer
:top-margin	integer (RSUs)
:bottom-margin	integer (RSUs)
:left-margin	integer (RSUs)
:right-margin	integer (RSUs)
:begin-new-page	t or nil
:begin-new-column	t or nil
:widow-count	integer
:straddle-columns	t or nil

To navigate within a table, there are a number of functions.

Function 14-3: get-parent
(tell *table* mid:get-parent)
Returns document

If you want to get move among rows, you can get the first or last rows of a table, and then use *get-next* or *get-previous*.

Function 14-4: get-child
(tell *table* mid:get-child)
Returns first row of table
Function 14-5: get-last-child
(tell *table* mid:get-last-child)
Returns last row of table

Now that you have a row, you probably want to get a cell. A cell is a child of a row:

Function 14–6: get–child

(tell *row* mid:get–child)

Returns first cell of table.

You can then use *get–next* and *get–previous* to move from cell to cell. But you can also use the following:

Function 14–7: get–cell

(tell *table* mid:get–cell *row column*)

Returns cell or microdocument at row number *row* and column number *column*

E.g. *(tell table mid:get–cell 0 0)*
Returns cell at row 0, column 0

Obviously, *get–cell* is zero-based.

If the cell only has one component, then *get–cell* returns the component. If it's a graphics frame (diagram) cell that contains only one component, then it returns the component; otherwise, it returns the graphics frame, then leaves it to you to navigate the frame objects.

Table totaler

The following function will total the contents of a column in a table. It is fairly limited. It skips cells that are set as frames, and it skips cells that have more than one component. It also doesn't do anything more interesting with the result than return it.

14–2 get–cmpn–number

```
(defun get–cmpn–number (cmpn)
; gets number in a cell
(let (m1 m2 contents number)
     (setq m1 (tell cmpn mid:get–marker)) ; beginning
     (setq m2 (tell cmpn mid:get–marker t)) ; end
     (setq contents (tell m1 mid:get–substring m2 t t))
     ; if contents, convert to float
     (if contents
       (setq number (atof contents))
       ; else, no contents, so return 0
       (setq number 0))
  number))
```

14–3 get–all–cmpns

```
(defun get–all–cmpns (cmpn)
; gets all cmpns in a cell and totals their contents
(let ( (total 0.0) number)
     (while cmpn
```

```
        ; get contents as a number
        (setq number (get-cmpn-number cmpn))
        (if number (setq total (+ total number)))
          (setq cmpn (tell cmpn mid:get-next))
      ) ; while

  total))
```

14–4 total–cells

```
(defun total-cells (table col-number)
(let ( (total 0.0) cell (row-ctr 1))
  ; loop, looking at every row
  (while (setq cell (tell table mid:get-cell row-ctr col-number))
    (setq total (+ total (get-all-cmpns cell)))
  ; increment row ctr
  (inc row-ctr)
  )
  total
))
```

Table sorter

The following sorts a table using the contents of a column specified by the user. It skips frame cells. And it should be used with care on tables that have vertically straddled cells.

14–5 table–sort–fcn

```
(defun table-sort-fcn (a b)
; used by sort as criterion
(let (conta contb ret)
  (setq conta (car a))
  (setq contb (car b))
  ; either nil?
  (if (and conta contb)
    (progn
      ; if it's a number string, convert to numbers
      (if (and
            (atof conta)
            (atof contb))
        (setq ret (< (atof conta) (atof contb)))
      ;else
      (progn
      (if (> (string-compare conta contb) 0)
        (setq ret t)
```

```
        ;else
        (setq ret nil))))))
        ; no content
        (if (or (not conta) (not contb))
          (setq ret t))
ret
))
```

14–6 get–cell–cont

```
(defun get–cell–cont (table col row)
; gets content of a cell in a particular row
(let (cell contents)
  (setq cell (tell table mid:get–cell row col))
  (if (is–of–class cell doc–cmpn–class)
    (progn
      (setq contents
        (tell (tell cell mid:get–marker) mid:get–substring
        (tell cell mid:get–marker t) t t))))
contents
))
```

14–7 sort–table

```
(defun sort–table ()
(let (rows cols sort–table r (ctr 1) col–to–sort cmpn item (row–ctr 1)
(row–list nil))

  ; is there a document open?
  (if (not (doc–current–icon))
    (progn
    (stk–open "No document is open.")
    (toplevel)))

  ; get row
  (setq r (doc–point–cmpn))
  (if (not (tell r mid:get–props :table–row))
    (progn
      (stk–open
        "You're either not in a table or not at the right level."
        "Get table caret and try again.")
      (toplevel)))

  ; get table
  (setq sort–table (tell r mid:get–parent))
```

```
(if (not (is-of-class sort-table doc-table-class))
  (progn
    (stk-open "Move caret to table and try again")
    (quit)))

; get number of rows and cols
(setq rows (tell sort-table mid:get-props :number-of-rows))
(setq cols (tell sort-table mid:get-props :number-of-columns))
; get col to sort on
(setq col-to-sort
(stk-open
  (format nil "Sort on which column (1-~D)" cols) :input 5) )
; error check
(if (not col-to-sort)
  (toplevel))
; convert input to ascii
(setq col-to-sort (atoi col-to-sort))
(if (not col-to-sort)
  (progn
    (stk-open "Improper (non-numeric) input. Try again.")
    (toplevel)))
(if (not (and (> col-to-sort 0) (<= col-to-sort cols)))
  (progn
    (stk-open (format nil "Column not in range. Choose 1-~D." cols))
      (toplevel)))

; get list of all rows: (content . row)
; go to top of table
(setq r (tell sort-table mid:get-child))
(while (and r
  (tell r mid:get-props :table-row)
  (equal sort-table (tell r mid:get-parent)))
; skip table heads
(if (not (tell r mid:get-props :table-header))
  (progn
    (push (cons (get-cell-cont sort-table col-to-sort row-ctr) r)
    row-list)))
(inc row-ctr)
(setq r (tell r mid:get-next)))

; sort the list
```

```
  (setq row-list (sort row-list 'table-sort-fcn))

  ; cut each member of list, go to beginning of table, and paste it
  ; go to end of table
  (doc-goto-marker (tell (get-last-row sort-table) mid:get-marker))
    (while (setq item (pop row-list))
    ; get cmpn
    (setq cmpn (cdr item))
    ; deselect all
    (tell *cmpn-editor* mid:deselect :all)
    ; select cmpn
    (tell *cmpn-editor* mid:select cmpn)
    ; cut cmpn
    (tell *cmpn-editor* mid:cut)
    (doc-goto-marker (tell (get-last-row sort-table) mid:get-marker))

    ; set caret direction
    (tell *cmpn-editor* mid:set-props :caret-direction :next)
    ; paste it
    (tell *cmpn-editor* mid:paste)

    (doc-goto-marker (tell cmpn mid:get-marker))

  )

  ; update the document
  (doc-flush-queue)
  ))
```

14–8 get–last–row

```
(defun get-last-row (table)
(let (r prev)
  ; get first
  (setq r (tell sort-table mid:get-child))
  ; get last
  (while (and r (equal table (tell r mid:get-parent)))
    (setq prev r)
    (setq r (tell r mid:get-next)))
    ; we went one past it, if there's something after the table
    ; (if r (setq r (tell r mid:get-previous)))
  prev
))
```

```
; bind it to ^XT
(kbd-bind kbd-doc-map "\^XT" 'sort-table)
```

Table striper

This next set of functions automatically add a background fill every nth row of a table; the user gets to select how many blank rows there ought to be between filled rows and also how many rows to fill. For example, you could have every fifth row filled, or have two filled, then three blank, etc.

Unfortunately, you can't tell what color the fill is going to be since you set colors by using the number of the color in the color palette for that document, and color palettes are very difficult to decode. So this arbitrarily uses the third color in the document's palette, whatever color that happens to be.

14–9 which-row

```
(defun which-row (row)
; returns number of row we're in ... zero based
(let ((row-ctr nil) table first-row)
  ; are we in a table at all?
  (if (equal 'doc-table (type-of (tell row mid:get-parent)))
    (progn
      (setq row-ctr 0)
      ;get the table
      (setq table (tell row mid:get-parent))
      ; get first row
      (setq first-row (tell table mid:get-child :along :structure))
      ; move up rows until row equals first row
      (while (not (equal row first-row))
        ; increase the row counter
        (inc row-ctr)
        ; move to previous row
        (setq row (tell row mid:get-previous)))))
 row-ctr
))
```

14–10 ts-get-table

```
(defun ts-get-table ()
 (let (rc-table)
  ; get table
  (setq rc-table (tell (doc-point-cmpn) mid:get-parent))
  ; is it a table?
  (if (not (typep rc-table 'doc-table))
    (progn
```

```
      (stk-open "You need to be in a table")
      (quit)))
rc-table
))
```

14–11 ts–get–cell

```
(defun ts-get-cell ( table r c )
; takes cell cmpn and returns the cell frame it's in
(let (contents)
  (setq contents (tell table mid:get-cell r c))
  (while (not (typep contents 'doc-frame))
   (setq contents (tell contents mid:get-parent)))
contents
))
```

14–12 stripe–a–row

```
(defun stripe-a-row (table row col-number fill-it)
; adds color to a row of a table
(let (cell (col-ctr 1) (color 3))
  (while (and (tell table mid:get-cell row col-number)
     (<= col-ctr col-number))
    ; get cell
    (setq cell (ts-get-cell table row col-ctr))
    ; set color props
    (tell cell mid:set-props :fill-color-index color)
    (tell cell mid:set-props :fill-visible fill-it)
    ; inc col-ctr
    (inc col-ctr))
))
```

14–13 do–stripes

```
(defun do-stripes ()
(let (table row-ctr (col-ctr 0) ts-col-number
  (rowson 0) (rowsoff 0) times r)
  ; get table
  (setq table (ts-get-table))
  ; get frequency of stripe
  (setq rowson (atoi (stk-open "Enter number of rows to stripe" :input 3)))
  (if rowson
    (setq rowsoff
    (atoi (stk-open "Enter number of rows to skip" :input 3)))
    ; else
    (quit))
  (if (= 0 (+ rowson rowsoff))
```

```
  (quit))

  ; get number of columns in table
  (setq ts-col-number (tell table mid:get-props :number-of-columns))
  ; loop through rows -- start striping at the caret
  (setq r (doc-point-cmpn))
   (setq row-ctr (1+ (which-row r)))

  (catch 'tdone
  (while r
    (repeat rowson
      ; stripe the row
      (stripe-a-row table row-ctr ts-col-number t)
      ; get next row
      (setq r (tell r mid:get-next ))
      ; inc rowctr
      (inc row-ctr)
      ; if over, then end
      (if (not r)
        (throw 'tdone))
    )
    (repeat rowsoff
    ; blank the row
    (stripe-a-row table row-ctr ts-col-number nil)
    ; get next row
    (setq r (tell r mid:get-next :along :structure))
    ; inc rowctr
    (inc row-ctr)
    ; if over, then end
    (if (not r)
      (throw 'tdone))
      )
))
(doc-flush-queue)
))
; try it
(do-stripes)
```

Chapter 15
Graphics

This chapter skims the surface of Interleaf graphics. Unfortunately, at the time of this writing, Interleaf Lisp's graphic capabilities are an advanced topic, requiring a lot of low-level coding. For the upcoming Motif version of Interleaf 5 Interleaf is developing a diagramming editor object which reportedly will be as easy to use as **text-editor**. Until then, the best this book can do is show you how to work with graphics using frames and named graphic objects (NGOs) — objects that can be created by the end user but manipulated by the developer.

Frames

To get the first frame in a document:

Function 15–1: get–first

(tell–class doc–frame–class mid:get–first &*optional* :along *axis)*

***Returns*: first frame**

E.g. *(tell–class doc–frame–class mid:get–first)*
Returns first frame

The two keywords are *:structure* and *:format.* Since the order of anchors in a document can be different from the order of the frames (e.g., a bottom frame's anchor may come before an at-anchor frame's anchor, but the bottom frame may come after the at-anchor frame in terms of the stream of objects on the page), which axis you navigate along makes a difference. For example:

(tell–class doc–frame–class mid:get–frame :along :structure)
Returns first frame in the order of the anchors in the document
(tell–class doc–frame–class mid:get–frame :along :format)
Returns first frame in the order of the frames in the document

Now that you have a frame, you can get the next or previous frame.

Function 15–2: get–next

(tell *frame* mid:get–next &*optional* :along *axis*)

Returns: next frame

E.g. *(tell myframe mid:get–next :along :name)*
Returns next frame after myframe with the same name as myframe

The axes are *:structure*, *:format* and *:name*. The structure axis will return the next frame in anchor order. (For repeating frames, it returns the next repeating frame on that page.) The format axis will return the next frame in frame order. The name axis returns the next frame with that name in random order; the name axis is handy for visiting every instance of a frame with a particular name.

Function 15–3: get–previous

This is exactly the same as *get–next*, discussed immediately above, except, of course, it moves backwards.

Remember, you can also use the various flavors of *doc–scan* to look at every frame. For example:

```
(doc–scan–class–apply 'do–something (list doc–frame–class))
Sends every frame to the function do–something
```

Do–something should have a single argument, which is the frame being examined, e.g.:

```
(defun do–something (fr)
  ; put the frame's name into a stickup
  (stk–open (tell fr mid:get–name))
  )
```

You can use the ever-elusive name-pool to look at every frame with a particular name.

15–1 frame–scan–name

```
(defun frame–scan–name (name fun)
; visits every frame named name and sends it to function fun
(let (pool head name inst)
  (when ; traverse name pool of named diagramming objects
    (setq pool (name–find–pool doc–frame–class))
    (setq head (tell pool mid:get–child))
        (while head  ; find all instances of name
          (setq inst (tell head mid:get–child :along :name))
            (while (and inst (string= (tell inst mid:get–name) name))
              (funcall fun inst)
                (setq inst (tell inst mid:get–next :along :name)) )
                (setq head (tell head mid:get–next))
  ) ; while head
)))
```

```
;try it out, with a do-something function presumably defined elsewhere
(frame-scan-name "rule" 'do-something)
```

This function takes the name of the frame and the function you want to send the frame to. So, you might use it to send every frame named "rule" to a function you write that sets its properties the way you want.

The following function will return the current selected frame. *Note: This function is used by several that follow.*

15–2 get-selected-frame

```
(defun get-selected-frame ()
(let (m1 m2 markers token)
  ; if nothing selected, quit
  (if (not (setq markers (doc-text-selection)))
    (progn
      (stk-open "Nothing selected")
      (quit)))
  ; get markers at beginning and end of text selection
  (setq m1 (car markers))
  (setq m2 (cdr markers))
  ; get the frame token
  (setq token (car (tell m1 mid:get-substring m2 (list :frame))))
  ; token is in form (:frame . frame-object). Return only the frame-object
  (cdr token)
))
```

Note that the above function doesn't work if the text selection spans components. If more than one frame is selected, it will return the first one in anchor order. The order of the markers returned by *doc-text-selection* depends upon whether the user selects downwards or upwards.

Function 15–4: get-props

(tell *frame* mid:get-props &keyword *prop*)

Returns*: *prop* property of *frame

E.g.
```
(tell myframe mid:get-props)
;Returns all the properties of myframe:
(:name "myframe"
:force-hidden nil
:attributes nil
:placement :at-anchor
:shared-content t
:frame-selection t
:border-visible t
```

:auto–edit nil
:height 1228800
:width 7373280
:vertical–alignment
:bottom
:vertical–offset 0
:baseline –1228800)

These properties clearly match the names on the frame property sheet.

You can get the following properties of frames:

Property	Argument	Comment
:name	string	frame's name
:force–hidden	t or nil	force it to hide or not
:placement	:at-anchor, :underlay, etc,	
:shared-content	t or nil	
:frame–selection	t or nil	clicking on frame selects it
:border–visible	t or nil	frame border visible when frame selected
:auto–edit	t or nil	clicking on object in closed frame opens frame and edits object
:height	rsu's	
:width	rsu's	
:left-offset	rsu's	
:top-offset	rsu's	
:vertical-alignment	:bottom. :top, etc.	
:horizontal–alignment	:left, :centered	
:horizontal–reference	:page–without–margins, etc.	as on prop sheet
:vertical–reference	:page–with–margins, etc.	as on prop sheet
:repeat–begin	t or nil	begin repeating frame
:repeat–end	t or nil	end repeating frame
:repeat–same–page	t or nil	begin repeater on this page

To set properties:

Function 15–5: set–props

(tell *frame* mid:set–props &keyword *prop*)

Returns*: *prop* property of *frame

E.g. *(tell myframe mid:set–props :placement :underlay)*
Turns myframe into an underlay frame

You can use **text-editor** to create frames. And since frames can come in with content, you can do some powerful things very easily.

15–3 create–draft–underlay

```
(defun create-draft-underlay ()
(let (frame)
  ; create the frame
  (tell *text-editor* mid:create :frame "draft")
  ; use get-selected-frame to get the new frame
  (setq frame (get-selected-frame))
  ; set its props ... although could also do this by setting "draft's" master by
  ; hand
  (tell frame mid:set-props :placement :underlay
                :shared-content t
                :repeat-begin t
                :repeat-same-page t
)

))
```

This script expects that you've already made a master frame called "draft" that contains the content you want. (You could create this graphic through I-Lisp too, but it's *much* easier to do so using Interleaf 5's graphics tools than through any programming language.)

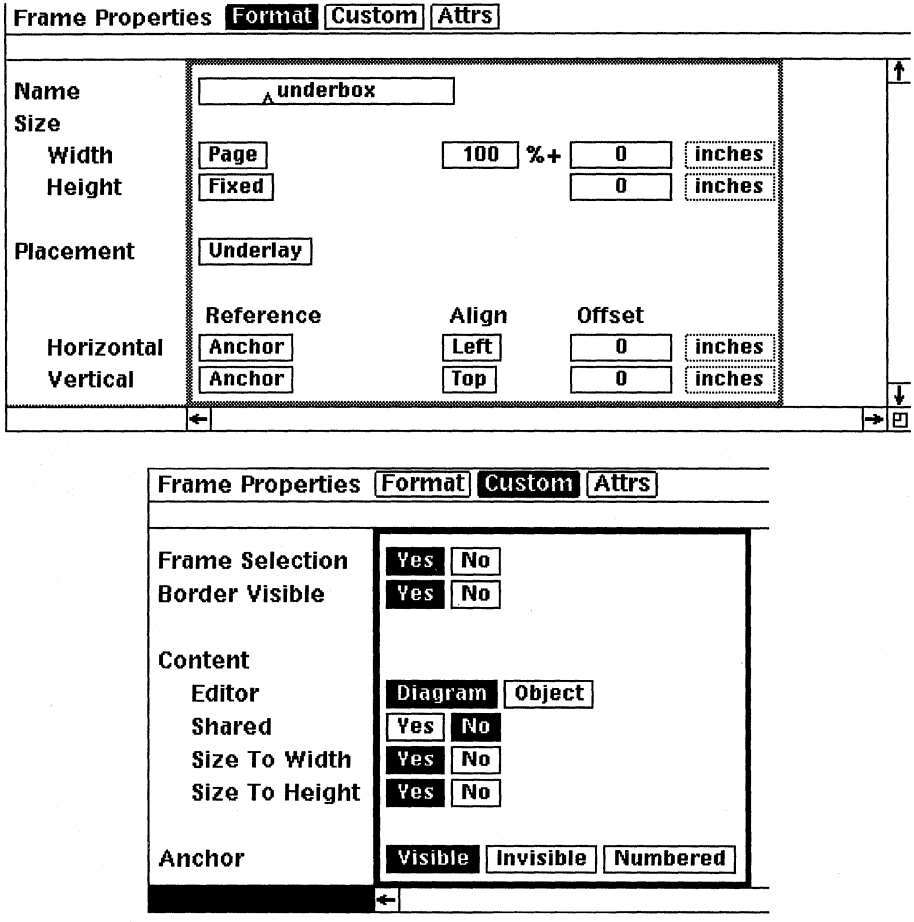
The following function will look for any frame named "draft" and will either hide it or show it, depending on which argument it's passed.

15–4 hide–draft–notice

```
(defun hide-draft-notice (name show)
(let (head m)
  (setq head (name-find-head doc-frame-class  name *document*))
  (if head
    (progn
      (setq m (tell head mid:get-child :along :name))
        (while m
          (tell m mid:set-props :force-hidden show)
          (setq m (tell m mid:get-next :along :name)))))
))

; bind ESC-h to hide frames named "draft"
(kbd-bind kbd-doc-esc-map "h" '(hide-draft-notice "draft" t))
; bind ESC-H to show frames named "draft"
(kbd-bind kbd-doc-esc-map "H" '(hide-draft-notice "draft" nil))
```

The following function sets the height and width properties of a frame so that they match the size of a component. This assumes you have used the user interface to create a master for an underlay frame named *underbox* which contains a box. The frame's properties should be set

to size contents to width and height. Here are two propsheets that show what it should look like.



Two "pages" of the underbox property sheet.

15–5 box–cmpn

```
(defun box–cmpn (cmpn)
; uses get–selected–frame, above
(let (height frame m1 m2 font font–size)
  ; get the height of the cmpn by getting diff. in position
  ; of beginning and end markers
  ; get beginning and end markers
  (setq m1 (tell cmpn mid:get–marker))
  (setq m2 (tell cmpn mid:get–marker t))
  ; get difference in height between them
  (setq height
    (– (cdr (tell m2 mid:get–bounds :type :screen))
    (cdr (tell m1 mid:get–bounds :type :screen))))
  ; if 0, then they're on the same line
  (if (= height 0)
    (progn
    ; make it equal to height of one line
    (setq font (tell cmpn mid:get–props :doc–font))
    (setq font–size (tell font mid:get–props :size))
    ; convert points to inches (72pts in an inch) to rsu's
     (setq height (round (* 1228800 (/ font–size 72.0))))
    )
    ; else – convert from pixels to inches to rsu's
    (setq height (round  (* 1228800 (/ height 75.0)))))
  ; go to beginning of cmpn
  (doc–goto–marker m1)
  ; create frame
  (tell *text–editor* mid:create :frame "underbox")
  ; frame is selected so use previous function to get it
  (setq frame (get–selected–frame))
  ; set its props
  (tell frame mid:set–props :height height)
))
```

Notice that this does a very bad job if the component is split across two pages. You could alter the function to take account of this, or you could only use it on components set so they won't split across pages.

The following function creates a drop cap like the one this paragraph begins with. It finds the first letter in a component, cuts it, creates a frame called "dropcap" (which you should have defined ahead of time), sticks the first letter into a microdocument in the frame, and adjusts the paragraph's initial indent. For this to work, the "dropcap" frame's master needs to have a single microdocument as its content, and you

should size the frame and position the way you want a dropped cap to work. Make sure the frame's properties are set so that its contents will size themselves to the frame's width to ensure that the microdocument displays properly.

15–6 drop–cap

```
(defun drop–cap ()
(let (m1 m2 text (c (doc–point–cmpn)) letter frame graph micro c cmpn)
  ; get first letter of cmpn
  (setq m1 (tell c mid:get–marker))
  ; find first letter
  (setq m2 (tell m1 mid:copy))
  (until (or letter
    (not  (zerop (tell m2 mid:move–by 1))))
      (setq letter (tell m1 mid:get–substring m2 (list :char :hyph–char)))
      (if (not letter)
    (setq m1 (tell m2 mid:copy))))
  ; delete the first character
  (tell m1 mid:delete m2)
  ; go to beginning of line
  (doc–goto–marker m1)
  ; create the frame
  (tell *text–editor* mid:create :frame "dropcap")
  ; get the frame using (get–selected–frame) defined in this chapter
  (setq frame (get–selected–frame))
  ; get dg–graph
  (tell frame mid:open)
  (setq graph (dg–remainder))
  ; get microdocument
  (setq micro (tell graph mid:get–child))
  ; get first (and only) cmpn in microdoc
  (setq cmpn (tell micro mid:get–child))
  ; insert the letter
  (tell (tell cmpn mid:get–marker t) mid:insert letter)
  ; adjust the initial indent to a half–inch, 3–line indent
  (tell c mid:set–props  :initial–indent 614400 :initial–indent–count 3)
  ; update the document
  (doc–flush–queue)
))
```

Architecture

Now let's take a look at the architecture of frames and diagrams. It's not what you might think from the user interface.

Every frame has a child of *dg–diagram–class*. This is the object — invisible to the end user — that gets changed when you turn the grid on or off, adjust gravity, etc.

```
(setq frame (get-selected-frame))
Returns selected frame, using function 15-3
(setq dg1 (tell frame mid:get-child))
Returns dg-diagram
(tell dg1 mid:get-props)
Returns
(:clipping-box ((0 . 0) (7356416 . 180224) -2147483646)
:default-fill-color 7
:default-fill-pattern 5
:default-fill-visible nil
:default-edge-color 7
:default-edge-dashes :none
:default-edge-weight 8
:default-edge-visible t
:default-text-color 7
:gravity t
:gravity-radius 1
:rotation-detent t
:rotation-detent-angle 15.0
 :magnified nil
:magnification-level 1.0
:width 8093696
:height 196608
:horizontal-shift 0
:vertical-shift 0
:input-scaled nil
:input-scale-factor 1.0
:old-text-angle 0.0
:grid-align t
:grid-displayed nil
:grid-on-top t
:grid-type :rectangular
:grid-horizontal-spacing 491520
:grid-horizontal-subdivisions 81920
:grid-vertical-spacing 491520
:grid-vertical-subdivisions 81920
```

:grid–isometric–spacing 491520
:grid–isometric–subdivisions 81920)

You can get the dg-diagram object of the frame currently being edited through:

Function 15–6: dg–graph
(dg–graph)
***Returns*: current diagramming object**

The contents of the frame are the children of this object. They are divided into two lists: those that are currently selected *(selection)* and those that are not *(remainder)*.

Function 15–7: dg–selection
(dg–selection)
***Returns*: list of selected diagramming objects**
Function 15–8: dg–remainder
(dg–remainder)
***Returns*: list of all unselected diagramming objects**

Now, for the reasons stated at the beginning of this chapter, we are not going to look at how to modify diagramming objects. But here is an example of what happens when you finally get to the props of a diagramming object.

```
(setq d (tell (dg–selection) mid:get–child))
Returns diagramming group
; get child of that diagramming group
(setq dd (tell d mid:get–child))
Returns actual diagramming object, e..g., ellipse
; get props of ellipse
(tell dd mid:get–props)
Returns (:locks nil :edge–color–index 7 :edge–dashes :none
:edge–weight 8 :edge–visible t :fill–color–index 7 :fill–pattern–index 5
:fill–visible nil)
```

Named Graphic Objects

With a little Lisp, you can do a lot with graphics, thanks to named graphics objects. These let you use the user interface to create complex graphics and then turn them into objects that can be manipulated through Lisp.

The following *doc–scan* variant visits each of the named graphic objects in a document and sends it to a function (in this case, a function called *ngo–test–fcn*). (Don't forget to *(require "doc–scan")* at the beginning of any script that uses this function.)

```
(defun ngo–test–fcn (o)
; dummy function for testing find–named–ngos
```

```
; puts name of NGO into stickup
  (stk-open (tell o mid:get-name)))
```

Here's the actual code you would run to visit every named graphic object:

15–7 find-all-ngos

```
(doc-scan-class-apply 'ngo-test-fcn (list dg-named-class))
```

This next function not only finds all NGOs and sends them to a function, but it will also optionally let you specify the name of the NGO you want to send to the function and/or the frame that you want to inspect. You tell it the name of the NGO you want to look at. You can optionally give it the name of a function you want every NGO sent to.

15–8 find-named-ngos

```
(defun find-named-ngos (name &optional fcn)
  (let (head ngo (ngo-list nil))
  (setq head (name-find-head dg-named-class name))
   (setq ngo (tell head mid:get-child))
  (while ngo
     (if fcn (funcall  fcn ngo))
    (push ngo ngo-list)
    (setq ngo (tell ngo mid:get-next :along :name)))
  ; return list of NGOs
  ngo-list
))
```

If you want to use this function to send every NGO to some other function, you design a function such as:

```
(defun dummy-code (ngo-object)
  ; do something here with the ngo-object
  )
```

You invoke the function either as *(find-named-ngos "my-ngo")* or *(find-named-ngos "my-ngo" 'dummy-code)*.

To make any further progress with graphics, I recommend you either poke around with documents you don't mind trashing during sessions you don't mind crashing, or wait for the next version of Interleaf 5 to provide a more accessible interface to the graphics system.

Chapter 16
Windows

If you are using the Interleaf user interface (and not, for example, Open Look or Motif), then this chapter will tell you how to access the Interleaf windowing system.

Window manager

Just as there are text editors and component editors, Interleaf provides an object that manages windows: **wn-wmgr**. Its properties include:

Property	Type	Comment
:pointer-window	object	Window the mouse pointer is in
:windows	list	List of all current window objects, in back to front order
:size	cons	Width and height of area that can hold windows
:position	cons	Upper left corner of where windows can be placed
:maximum-bounds	cons	Lower right corner where windows can be placed

The first two properties enable you to get a window object. For example,

```
(tell *wn-wmgr* mid:get-props :pointer-window)
Returns window object the mouse pointer is in
```

The window manager also has a set of methods.

Function 16-1: refresh

(tell *wn-wmgr* mid:refresh)

Redraws all windows

This does the same thing as doing a refresh through the desktop popup.

Function 16-2: set-props

(tell *wn-wmgr* mid:set-props *keyword value*)

Sets *property keyword* to *value*

E.g. *(tell *wn–wmgr* mid:set–props :left–button :select :middle–button :menu :right–button :extend)*
Returns :extend

There aren't many properties you can set for the window manager. The only one you might need is the ability to set which mouse button does what, as in the example immediately above.

If you want to get a particular window and know its name, you can use one of the properties of window objects:

Function 16–3: get–object

(tell *window* mid:get–object)

Returns*: object of *window

If the window is a document window, then this will get the document editor. If it's a directory window, it will get the desktop icon. The following function allows you to find a window by supplying the name of the object you're looking for, document or container.

16–1 find–window

```
(defun find–window (name)
; finds window of a particular name
  (let (win–list win obj parent–name target–obj)
    ; get list of windows
    (setq win–list (tell *wn–wmgr* mid:get–props :windows))
    ; loop through list looking for name
    (while (setq win (pop win–list))
      ; get the object associated with window
      (setq obj (tell win mid:get–object))
      ; is there an object at all?
      (if obj
        (progn
          ; is it a document?
          (if (is–of–class obj doc–editor–class)
            (progn
              (setq parent–name (tell (tell obj mid:get–object)
              mid:get–name)))
            ; else it's a container
            (progn
              (setq parent–name (tell obj mid:get–name))))
          ; if it's the right name, save the object
          (if (string= name parent–name)
              (setq target–obj (tell obj mid:get–window))))
    ))
```

```
  target-obj
  ))
```

Now that you have a window, what can you do with it? You can get and set some properties:

Property	Type	Comment
:position	pixels, cons	x and y position
:size	pixels, cons	height and width
:interior-position	pixels, cons	height and width
:interior-size	pixels, cons	height and width
:object	desktop object	object it's a window of
:obscured	t or nil	visible or not
:pointer-position	pixels, cons	x and y position

There are other functions as well:

Function 16–4: new

(tell win mid:new)

This creates a new window.

Function 16–5: close

(tell win mid:close)

Closes the window.

Function 16–6: button

(tell win mid:button *button state area*)

This allows you to intercept mouse button presses and do what you want with them. The buttons are *:menu, :popup* or *:select.* The state is either *:up* or *:down.* The area is one of *:command, :message, :resize, :title* or *:window*, corresponding to various areas of the window.

This allows you to get the popup you'd get in a window without actually being in that window:

16–2 get-win-popup

```
(defun get-win-popup (win)
  (tell win mid:button :menu :down :message)
)
```

You could also provide your window with a new *button* method so that you can give it entirely new popups:

16–3 new-win-popup

```
; Assumes you already have a window setq'ed to w
; create new win class
```

```
(setq new-win-class (obj-new-class wn-window-class "New Win"))
; give your win class a new button method.
(obj-provide new-win-class mid:button 'new-win-pup-handler)
; make your window a member of that class
(tell w mid:set-class new-win-class)
```

16–4 new-win-pup-handler

```
(defun new-win-pup-handler (win button state area)
(let (pup)
  (setq pup
    (popup-create-list
      (popup-create-entry "New function" 'new-pup-fcn)))
    (if (and (eql button :menu)
        (eql state :down)
        (eql area :message))
        (progn
          (popup-run pup)))
))

(defun new-pup-fcn (data)
; dummy function for popup handler
  (stk-open"Dummy function"))
```

This creates a new class of window. If you press the menu button *while in the message area of the window* (the line under the title, usually), you will now get a single-entry popup that says "New function" and which executes a dummy function.

There are other properties of windows you can set.

Function 16–7: lower-to-bottom

(tell win mid:lower-to-bottom)

Lowers window to bottom, most obscured position on screen.

Function 16–8: raise-to-top

(tell win mid:raise-to-top)

Raises window to top on screen.

Function 16–9: message

(tell win mid:message *text*)

Displays text in message bar of window.

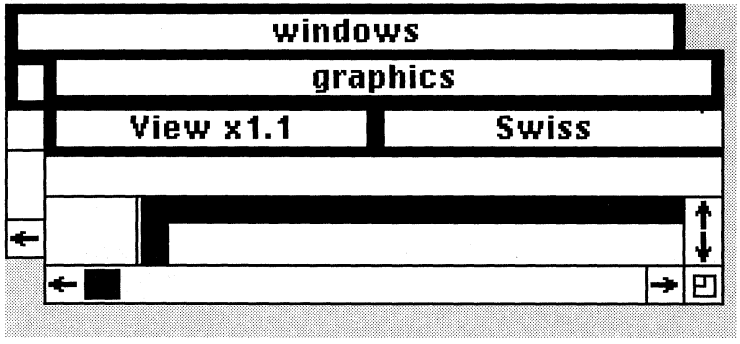
This does the same as *wn-message*. *Wn-message* automatically does a format on the text you supply. As with format, it expects a nil before the message. Here's an example:

```
(wn-message nil "Your name is ~A and your age is ~D" "Bill" 15)
```

Puts "Your name is Bill and your age is 15" into the window message bar.

An example

The following example lets you shrink a window so that only its name box (plus a tad more) shows, and stacks these "icons" in a diagonal line, starting from the upper left of your screen. Another keystroke restores the document to full-screen size. This is especially useful on small screens. This script should go into your profile drawer.



Two "iconized" windows, shown actual size.

```
; Compute size of desktop
(setq *dt-ht* (sys-get-vars :screen-height))
(setq *dt-wd* (sys-get-vars :screen-width))
```

16-5 rsus-to-pixels

```
(defun rsus-to-pixels (r)
; general utility for converting rsu's to screen pixels. Approximate
 (round (* (/ r rsus-per-inch) 75))
)
```

16-6 window-size-to-page

```
(defun window-size-to-page ()
; Sizes window to size of page
 (let (ph pw ht-offset wd-offset win zoom-factor)
  ; get the window
  (setq win (tell *doc-editor* mid:get-window))
  ; size of window over size of doc
  (setq ht-offset 150)
  (setq wd-offset 130)
  ;get doc ht and width
  (setq pw (tell *document* mid:get-props :page-width))
  (setq ph (tell *document* mid:get-props :page-height))
```

```
  ; convert from rsus into pixels
  (setq pw (rsus-to-pixels pw))
  (setq ph (rsus-to-pixels ph))
  ; get zoom factor
  (setq zoom-factor (tell *doc-editor* mid:get-props :zoom))
  ; get visible page width and height (multiplying page size by zoom)
  (setq pw (round (* pw zoom-factor)))
  (setq ph (round (* ph zoom-factor)))
  ; add size of windowing stuff
  (setq pw (+ pw wd-offset))
  (setq ph (+ ph ht-offset))
  ; if greater than desktop size, reduce to desktop size
  (if (> ph *dt-ht*)
    (progn
      (setq ph (1- *dt-ht*))
      (tell win mid:set-props
        :position (cons (car (tell win mid:get-props :position)) 0
    ))))
  (if (> pw *dt-wd*)
    (progn
      (setq pw (1- *dt-wd*))
      (tell win mid:set-props :position (cons 0
    (cdr (tell win mid:get-props :position))
    ))))
  ; set window size
  (tell win mid:set-props :size (cons pw ph))

  ; put page squarely on screen, without visible page break
  (tell *doc-editor* mid:set-props :page
  (tell *doc-editor* mid:get-props :page))

  ; raise window to top
  (tell win mid:raise-to-top)
))
```

16–7 grow-window

```
(defun grow-window ()
 ; grows window to max size, and brings it to the top
(let (win pos)
  (setq win (tell *doc-editor* mid:get-window))  ; get the window
  ; did we already save the doc's old window position?
  (setq pos (tell *document* mid:get-data :window-pos))
  ; If haven't already handled this window, so there's no pos data
```

```
  (if (not pos)
    (setq pos (cons 0 0)))
  ; set the position
  (tell win mid:set-props :position pos)
  ; size the window to the size of the page
  (window-size-to-page)
  ; raise window to top
  (tell win mid:raise-to-top)
))
```

16–8 this-is–small

```
(defun this-is-small (w)
 ; is this window a small one? Judges solely on size of window.
  (equal (cons *small-wd* *small-ht*)
        (tell w mid:get-props :size))
)
```

16–9 shrink–windows

```
(defun shrink-window ()
 ; shrinks window and reshuffles all small windows for semi-pleasing
 ;display
(let ((small-wins nil) win obj item wins lr tb
    (left-offset 15) (tb-offset 18))
  ; get the window of this doc
  (setq win (tell *doc-editor* mid:get-window))
  ; is this itself a small win? Assumes it is if it's the size of the small
  ; windows
  ; get all the windows
  (setq wins (tell *wn-wmgr* mid:get-props :windows))
  ; find any other docs and get furthest left and lowest
  (while (setq item (pop wins))  ; cycle through windows
    (setq obj (tell item mid:get-object))
    ; is it a doc window already reduced by this program?
    (if (and obj
      (is-of-class obj doc-editor-class)
      (this-is-small item))
      (progn
        (push item small-wins))))
        ; sort the list by top position
        (setq small-wins (sort small-wins #'sort-win-pos))
        ; place windows
        (setq lr 0) (setq tb 0)
        (while (setq item (pop small-wins))
```

```
        (tell item mid:set-props
          :position (cons lr tb) :size (cons *small-wd* *small-ht*))
        (tell item mid:raise-to-top)
        (setq lr (+ lr left-offset))
        (setq tb (+ tb tb-offset))
      )
    ; place this doc's window unless it started as a small one
    (unless (this-is-small win)
    ; save origins
    (tell *document* mid:put-data :window-pos (tell win mid:get-props
    :position))
    (tell win mid:set-props
      :position (cons lr tb)
      :size (cons *small-wd* *small-ht*))
  (tell win mid:raise-to-top))
))
```

16–10 sort-win-pos

```
(defun sort-win-pos (a b)
 ; sorts windows by vertical placement
(let (a1 a2)
  (setq a1 (car (tell a mid:get-props :position)))
  (setq a2 (car (tell b mid:get-props :position)))
  (< a1 a2)
  ))

; bind the keys
(kbd-bind kbd-doc-map "\^Xi" 'shrink-window)
(kbd-bind kbd-doc-map "\^XI" 'grow-window)
(kbd-bind kbd-doc-map "\^Xz" 'window-size-to-page)
```

Timers

A timer is a stop watch set by the system on your command. For some reason, timers are attached to windows.

Timers have the following properties:

Property	Comment
:delta–time	Number of milliseconds the timer lasts
:idle–time	Number of milliseconds the timer can be left idle before it kills itself
:repeat	If t, time restarts itself automatically
:active	t if timer has started and hasn't died yet

The methods *:new*, *:start* and *:stop* are self–explanatory. The following one isn't:

Function 16–10: end

(tell timer mid:end 'function)

Evaluates function when the time is up

If you use this method, when your timer expires, it will execute the function you name. If you do not set this, it will default to evaluating whatever function (if any) you've specified by putting data named :expression on the timer. For example:

```
(tell my–timer mid:put–data :expression '(blow–up–time–bomb))
```

If you now started my–timer (remember that you first have to create it, of course), when it expired it would run your function blow–up–time–bomb.

Here's an example of a function that prompts you to enter a number of minutes, and which puts up a stickup when that time has expired.

16–11 stkup–countdown–timer

```
(defun stkup–countdown–timer ()
  (let (minutes millisecs timer msg)
  (setq minutes (stk–open "How many minutes to count down?" :input 5))
  (if (string= "" minutes)
    (quit))
  ; convert to milliseconds
  (setq millisecs (* 60000 (atoi minutes)))
  ; create new timer
  (setq timer (tell–class wn–timer–class mid:new))
  ; tell it what to do when it expires and input a message
  ; (the comma before "timer" isn't a typo, but will remain a mystery in this
  ; book)
  (tell timer mid:put–data :expression '(countdown–done ,timer))
  ; get message to put into stickup and attach it to timer
  (setq msg (stk–open "msg " :input 30))
  (tell timer mid:put–data :text msg)
  ; set timer props
  (tell timer mid:set–props :delta–time millisecs)
  ; start the timer
  (tell timer mid:start)))
```

16–12 countdown–done

```
(defun countdown–done (timer–obj)
; what happens when the timer is done
  (stk–open (format nil "~D minute timer done.\n~A"
    (itoa (/ (tell timer–obj mid:get–props :delta–time) 60000))
    (tell timer–obj mid:get–data :text))))
```

```
; try it out
(stkup-countdown-timer)
```

It's easy to think of ways this could be elaborated. You could tell the user the current time in the stickup and have her enter the time for the alarm to go off. You could alter the interface, perhaps using a graphic of a clock to let the user set the time for the alarm. The actions that can occur can go far beyond opening a stickup. (The function *(beep)* works on some systems.)

Here's a little typing test. (Since it doesn't check for accuracy, it counts speed without testing skill.)

16–13 typing–test

```
(defun typing-test ()
  (let (cmpn timer text)
  ; create text to type
  (setq text "Now is the time for all good human beings to come to the aid
  of their party, except for lazy dogs too busy to jump over quick brown
  foxes. A man, a plan, a canal, panama. If you're done, then type it all
  again. And, please, drive carefully!")
  (stk-open "This will test your typing speed. When ready, press the button
    and type in the text you'll find inserted into your document.")
  ; create the components we need
  (setq cmpn (tell *cmpn-editor* mid:create "para"))
  (tell (tell cmpn mid:get-marker) mid:insert text)
  ; create blank para for typing into
  (setq cmpn (tell *cmpn-editor* mid:create "para"))
  ; start the timer for one minute
  (setq timer (tell-class wn-timer-class mid:new))
  ; tell it what to do when it expires
  (tell timer mid:put-data :expression '(typetest-done))
  ; set timer props to one minute
  (tell timer mid:set-props :delta-time 60000)
  ; start the timer
  (tell timer mid:start)))
```

16–14 typetest–done

```
(defun typetest-done ()
; count the words
  (let (word-count)
  ; get word count
  (setq word-count (count-words-in-cmpn (doc-point-cmpn)))
  (stk-open (format nil "You've typed ~D words in a minute, many of them
```

```
perhaps correctly." word-count))
  ))
```

16–15 count-words-in-cmpn

```
(defun count-words-in-cmpn (cmpn)
  (let (m (ctr 0))
    ; get marker at beginning of cmpn
    (setq m (tell cmpn mid:get-marker))
    ; loop until no words left
    (while (zerop (tell m mid:move-by 1 :by :word-endings))
      ; increment counter
      (inc ctr))
  ; return the ctr
  ctr
  ))
; try it
(typing-test)
```

Chapter 17
Streams

Streams in Interleaf Lisp allow you to access information in files outside of the Interleaf desktop. A stream is a data source, either coming into Interleaf or going out. In this chapter, we'll cover the basics of reading and writing files.

To use a stream, first you must *open* it.

Function 17–1: open

(open *filename*)

Returns*: stream to *filename

E.g. *(open "desktop/System5.cab/readme")*
Opens "desktop/System5.cab/readme"

Open takes a number of keywords. At a minimum, you'll want to specify *:input*, *:output* or *:io* (both), where *input* means you'll be getting input from it (i.e., reading it) and *output* means you'll be writing to it. You'll also want to specify either *:character* or *:byte* as the type of data you'll be inputting or outputting, although within the scope of this book it will almost certainly be characters that you'll be dealing with; you look for characters in text files and bytes in compiled, executable files (among others). If you have specified *:output* or *:io*, then you might want to specify either *:overwrite* (i.e., delete the existing material in the file when you write to it) or *:append*, so that the new material is appended to the end of the file. Finally, you can say what you want to have happen if the file does not exist and you are trying to write to it; *:create* will cause a file to be created.

Here are some examples:

```
(open "myfile" :input :character)
```
Opens myfile to read characters from (assumes "myfile" exists)
```
(open "myfile" :output :character :append :create)
```
Opens myfile for writing characters to. New data will be appended to the old. If no file by that name exists, a new one will be created.

Function 17–2: close

(close *stream*)

***Returns*: nil**

Close closes the stream. If you don't do this, some or all of the changes you made to the file may not be preserved.

Here is an example of opening (or creating a file), writing something to it, and closing it.

```
(let (out-file)
  ; look for file foo or create a new one if none there. Make it for overwriting
  ; characters to
  (setq out-file (open "foo" :output :character :overwrite :create))
  ; write a line to out-file
  (write-line "Howdy!" out-file)
  ; close it
  (close out-file)
  )
```

Reading files

Now let's look at a few ways of reading a file.

Function 17–3: read-line

(read-line *stream*)

***Returns*: characters from stream up to but not including the next hard return**

If you loop through a file using *read-line*, you will get every line in the file. When there are no more lines, you will get an *eof* (end-of-file) symbol. So, you want to loop until you get *eof*. For example:

17–1 example-read-loop

```
(defun example-read-loop (my-stream)
(let (line ctr)
  ; for this example, we'll just count lines
  (setq ctr 0)
  ; read lines until one equals the end of file marker
  (until (eql (setq line (read-line my-stream)) 'eof)
    ; do something with the line, such as increase a counter
    (inc ctr))
))
```

If your file isn't arranged in lines, you can specify the size of the bite you want to take out of the file:

Function 17–4: read–string–no–hang

(read–string–no–hang *number stream*)

Returns: *number* of characters from *stream*

E.g. *(read–string–no–hang 32 my–stream)*
Reads 32 characters from the stream my–stream

There is another, similar, function called *read–string*, but it doesn't know enough to return *eof* when the file is empty, so *read–string–no–hang* is almost always preferable for the sort of work we adventurers do.

Function 17–5: read–delimited–string

(read–delimited–string *delimiter stream*)

Returns: string from *stream* up to and including the *delimiter* character

E.g. *(read–delimited–string #\t my–stream)*
Reads all characters up to and including the next tab in my–stream

Read–delimited–string is useful where you know that your file uses something other than hard returns to divide its data. Also, if you use *read–delimited–string* and give it a hard return as its delimiter (#\n), it will include that hard return in the line that it gives back to you. (Beware: if the last line of the file doesn't end with a hard return, this function will not automatically put one in for you.)

Function 17–6: read–char–no–hang

(read–char–no-hang *stream*)

Returns: next character from *stream* or nil if at the end of the file

Read–char–no–hang is useful where you want to read a file character by character. Notice that it returns nil and not *eof* when it comes to the end of the file.

Function 17–7: unread–char

(unread–char *stream*)

Returns: puts the most recently read character back into *stream*

You will be surprised to find how useful this function is. Basically, it lets you look at a character and then put it back into the stream so that the next time you read the stream, you'll get the character.

Writing to files

Function 17–8: princ

(princ *string stream*)

Prints *string* to the *stream*

Princ is the function to use if you are trying to write a file that will be readable by humans, with no special objects. If you want to write something readable by Interleaf Lisp, use:

Function 17–9: prin1

(prin1 *string stream*)

Prints *string* to *stream*

For example, let's make an example file named "testfile" with a stream *setq*ed to a variable *f*.

```
; open a file for writing out to
(setq f (open "testfile" :output :character :append :create))
```

Here are the results of the various ways of using *princ* and *prin1*

Function	Result
(prin1 "line written with prin1" f)	"line written with prin1"
(princ (concat "Here's a princ character: " #\a) f)	Here's a princ character: a
(prin1 (concat "Here's a prin1 character: " #\a) f)	"Here's a prin1 character: a"
(princ (list 1 2) f)	(1 2)
(prin1 (list 1 2) f)	(1 2)

Now don't forget to close the file: *(close f)*.

Function 17–10: terpri

(terpri *stream)*

***Prints hard return into* stream**

This simply puts a hard return into the stream

There are also functions that correspond to the functions for reading files.

Function 17–11: write–line

(write–line *string stream*)

***Writes* string *to* stream**

Function 17–12: write–string

(write–string *string stream*)

***Writes* string *to* stream**

The key difference between these two is that *write–line*, unsurprisingly, puts a hard return after each line it writes.

Both of these functions can take arguments which specify which part of the string is to be written. For example, *(write–string "abcdef" out–stream :start 1 :end 3)* would write "bcd" to the out–stream file. (Yes, it's zero-based.)

Function 17–13: write–char

(write–char *char stream*)

Writes *char* to *stream*

Examples

This first example reads a file and applies the data in it to a chart. We have not discussed charts in this book, and this example is really a way of sneaking some chart Lisp in.

The program assumes that the data file looks a lot like the chart data sheet in Interleaf. It consists of lines of numbers, each number separated by a tab. (It also assumes that you've already gotten your hands on the chart object, which you can do using any of the standard techniques, looking for *dg–chart–class* objects.)

The way it works is simple. The function *update–chart* reads the file a line at a time. It passes each line to *get–row–of–values* which transforms it from a set of numbers separated by tabs into a list of numbers. But charts want their data as a list of entries, each of which has the row number, the column number, and the value. For example,

((0 0 1.0) (0 1 2.0) (1 0 5.0) (1 1 6.6) (2 0 7.0) (2 1 8.8))

This translates to:

	Column 0	Column 1
Row 0	1.0	2.0
Row 1	5.0	6.6
Row 2	7.0	8.8

Also, you may notice that these two functions share a variable: *chart–data–list*. This is an example of dynamic scoping, which is an advanced topic we will not cover except to say that because Interleaf Lisp allows for dynamic scoping, a function called by another function has access to that function's variables.

17–2 get–row–of–values

```
(defun get–row–of–values (line row)
; takes tab–delimited row and returns list of numbers
(let (n  (pos 0) (col 0) (prev–pos 0))
  ; find all the numbers by looking for tabs starting at where previous tab
  ; was found
  (while (setq pos (string–contained "\t" line prev–pos))
    ; get the characters that are the number and convert to a float
    (setq n (atof (substring line prev–pos pos)))
    ; pust (row col number) on to the chart–data–list
    ; (this is an example of dynamic scoping)
    (push (list row  col n) chart–data–list)
    (inc col)
    (setq prev–pos (1+ pos))
  )
```

```
  ; get the last number in the row
  (setq n (atof (substring line prev-pos (length line))))
  (push (list  row col n) chart-data-list)
  ; check for longest row so we can set size of chart
  (inc col) ; col is zero-based but chart size isn't, so boost col number
  (if (> col *chart-longest-row*)
    (setq *chart-longest-row* col))
))
```

17–3 update–chart

```
(defun update-chart (chart file-name)
(let (data-file line number (chart-data-list nil) (row 0))
   ; look for file ... quit if it's not there
  (if (not (probe-file file-name))
    (progn
    (stk-open (format nil "Cannot find ~A" file-name))
    (quit)))

  ; prepare to find longest row, using a global
  (setq *chart-longest-row* 0)

   ; open the file
  (setq data-file (open file-name :input :character))
  ; read file line by line
  (while (not (eql 'eof (setq line (read-line data-file))))
    ; push row, col and list of numbers on to larger list
    ;  (setq chart-data-list
    (get-row-of-values line row )
    ; increment row because we finished a line
    (inc row))
  (close data-file)

  ; do chart work
  ; set number of rows and cols; :num-vars is number of cols, num-vals is
  ; rows
  (tell chart mid:set-props :num-vals row :num-vars
*chart-longest-row*)
  ; set the chart values
  (tell chart mid:set-values chart-data-list)
  ; tell chart to update itself
  (tell chart mid:draw)
 ))
```

Remember, to use this function, you must already have access to a chart and know the pathname of the data file.

The following function does something quite different. It allows you to attach notes to icons so that if you type ^N while on the desktop, you'll be shown whatever note you attached (in a stickup) and will be able to alter it. That piece of the functionality is actually quite briefly accomplished. The rest of the function writes the pathname of the document, the date, and the note to a file in your system cabinet. It actually checks to see if there is already an entry for that file and overwrites it. (Actually, it copies each line of the file into a new file, replacing the old entry with a new entry if required; then it deletes the old file and renames the new one.) From an end-user point of view, the existence of this simple file allows you to find misplaced documents by doing a search for keywords in the notes.

The notes are stored as an attribute to the icon. We could save it as a comma-6 file (through *mid:put–saved–data*), but that would make a new file for each document that has a note, which seems inefficient.

17–4 dt–note–it

```
(defun dt–note–it ( )
; get or show a note for a selected icon
(let (note stk–text new–note icon)
  ; make sure the current container is the one the mouse is in
  (dt–set–container (dt–pointer–container))
  (setq icon (dt–child–selected))
  ; build text for stickup, showing the selected doc's name
  (setq stk–text  (format nil "Note for \^?~A\"" (tell icon mid:get–name)))
  ; get the current note to put in as a prompt in the stickup
  (setq note (tell icon mid:get–attrs "dt–note"))
  ; open the stickup
  (setq new–note (stk–open–prompt stk–text  :initial–prompt note))
  ; if user enters a new note
  (if new–note
    (progn
      ; put the note in as an attribute
      (tell icon mid:set–attrs "dt–note" new–note)
      ; update log
      (dtnote–update–log (list (dt–path icon :root *dt–desktop*)
      new–note))
  ))
  note
))
```

17–5 dtnote–get–date

```
(defun dtnote–get–date ()
; see explanation after the code
(strftime "%B %d, %Y")
)
```

17–6 dtnote–update–log

```
(defun dtnote–update–log (data)
; update the log we keep of all notes
(let (log–stream note–log–path filename note  date line)
  ; get the date string
  (setq date (dtnote–get–date))
  (setq filename (first data))
  (setq note (second data))
  ; build the line to be written into the log file
  (setq line (format nil "~A\t~A\t~A" filename date note))
  ; get path to system cabinet
  (setq system–path (dt–path *dt–system* ))
  ; make file name ... UNIX only! Comment out this or the DOS line below.
  ;(setq note–log–path (concat system–path "/noteslog"))
   ; here's the DOS version:
  (setq note–log–path (concat system–path "\\noteslog"))
  ; look for it in profile
  (if (not (probe–file note–log–path))
    (progn
    ; wasn't a log file. Ask user if she wants to make one
     (if (stk–open "No dt–note log file found in System cabinet. Make one?"
    :yes–no)
      (progn
        ; open a stream
        (setq log–stream (open note–log–path :output :character :append
        :create))
        ; enter initial warning text and some hard returns
        (princ "Created automatically by dt–note.lsp. Do not alter."
        log–stream)
        ; put in two hard returns
        (terpri log–stream) (terpri log–stream)
        ; put in the line of text we want
        (princ line log–stream)
        (close log–stream)
        )))
      ; else already is a notesfile, so go to the following function
      (progn
```

```
        (dtnote-add-line note-log-path filename line)))
))

(defun dtnote-add-line (path filename text)
; checks notefile for other entries for this filename. If it finds none,
; it appends the new text. If it finds one, it replaces it. Does this
; by copying each line of current into a temp file, then renaming
; the temp file at the end
(let (line in-stream out-stream tmp-path beheaded-path old-file)
  ; open current notes file to read from
  (setq in-stream (open path :input :character))
  ; create new file to copy into
  ; first get path of current dtnotes file, minus file name
  (setq beheaded-path (car (path-split path)))
  ; create temporary file using that pathname
  (setq tmp-path (get-temp-file beheaded-path))
  (setq out-stream (open tmp-path :output :character))
  ; read through current file line by line
  (until (eql (setq line (read-line in-stream)) 'eof)
    ; is it a line describing same file?
    (if (and (string-contained filename line)
             (= 0 (string-contained filename line)))
      (progn
      (write-line text out-stream)
      ; flag that we found an entry already
      (setq old-file t))
    ; otherwise
    (progn
      (write-line line out-stream)))
  )
  ; if not already an entry, make it one
  (if (not old-file)
    (write-line text out-stream))
  ; close the streams
  (close out-stream)
  (close in-stream)
  ; delete the current notes file
  (delete-file path)
  ; rename the temp file to the notes file name
  (rename-file tmp-path path)
))
```

```
; rebind it to ^N on the destop
(kbd-bind kbd-dt-map "\^N" 'dt-note-it)
```

This uses *strftime* to provide a formatted date string.

Function 17–14: strftime

(strftime *format*)

***Returns*: date string**

E.g. *(strftime "%B %d, %Y")*
Returns date in form "month day, year," e.g., "February 15, 1993)"

The formatting string can contain a wide number of variables representing various times and dates:

Formatting variable	Result	Example
%a	Abbreviated weekday name	Mon
%A	Full weekday name	Monday
%b	Abbreviated month name	Feb
%B	Full month name	February
%c	One particular date format	Monday, February 15, 1993 4:12 pm
%d	Day of the month as number	15
%H	24-hour time	16 (= 4pm)
%I	12-hour time	4
%j	Day of the year	46
%m	Month as number	2
%M	Minute	31
%p	AM or PM	pm
%S	Second	50
%U	Week number	6 (Sunday as zero)
%w	Weekday number	1 (Sunday is 0)
%W	Week number	6 (Monday as zero)
%x	One particular date format	Monday, February 15, 1993
%X	One particular time format	4:34 pm
%y	Year without century	93
%Y	Year	1993
%Z	Timezone	EST

Chapter 18
Active Documents

Now that you know how to build some Lisp scripts, if you attach them to a document, you've built yourself an active document.

Architecting and opening

Active documents can be architected in many ways. The task is to load some Lisp automatically so that the document knows that it is of a new class, and then to load some more Lisp telling the new class of document how to behave.

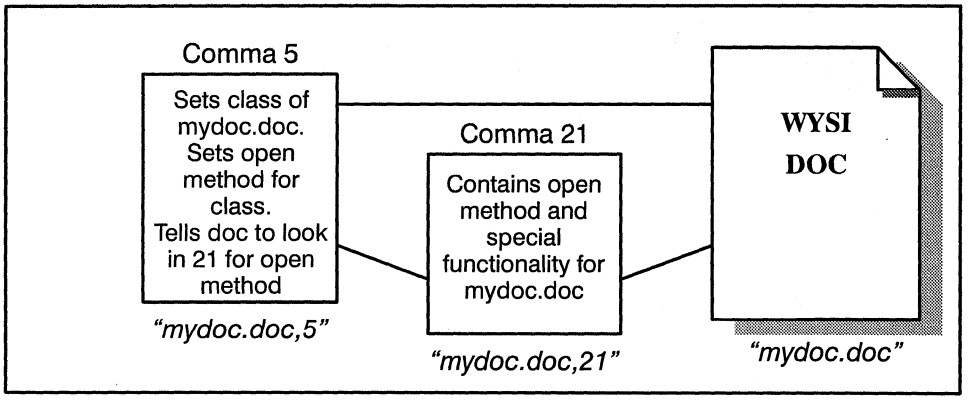
Typically, this is done through part files — files invisible to the end user but managed by the desktop. One special part file, the comma 5 file, is used to reclass the document because the comma 5 file is loaded automatically the first time that the directory the document is in is opened. So, when you open a folder (or book, etc.) for the first time, the system checks to see if there are any comma 5 files. If there are, they get loaded. (The comma 5 file for a document named *foo.doc* would be named *foo.doc,5*. In DOS, it would be named *foo.d#5*, and the comma 5 file for a folder named *barfld* would be *barfld.#5*.)

Let's take a look at a typical architecture for a document named *calc.doc*

- The comma 5 file *calc.doc,5* gets read when the directory containing *calc.doc* first opens. It creates a new class (e.g., *calc–class*), reclasses *calc.doc* to *calc–class*, gives *calc–class* a new open method, and tells *calc.doc* to look in a comma 21 file to find the new open method.
- The comma 21 file *calc.doc,21* (in DOS, *calc.d#l*) contains a new open method that, in addition to opening the document, perhaps creates a new class of text editor, builds some new popups, gathers data from various sources, etc.

While the comma 5 file gets read when the container is first opened (and isn't read again unless you explicitly force it to), the comma 21 file gets read when the document opens. This is because the comma 5 file typically tells the system to look in the comma 21 file to

find the document's open method. (You can specify a part file from 21-26.)



Let's take a look at a complete active document. In this example, we'll build a document that provides some basic address book functionality, for alphabetizing entries and inserting dividers between letters.

Comma 5 file

```
; unless we've already created the new class, create a new class of doc as
; a subclass of the generic document class
(unless (boundp 'addr-bk-class)
  (setq addr-bk-class
    (obj-new-class dt-document-class "Address Bk"))

  ; tell it to look in its comma 21 file to find its new open method
  (defautoload addrbk-open
    (dt-path *dt-load-object* :part 21))
)

; give the addr-bk-class a new open method, a function called
; addrbk-open
(obj-provide addr-bk-class mid:open 'addrbk-open)
; tell this document to set its class to the new class
(tell *dt-load-object* mid:set-class addr-bk-class)
```

You can use this example as a template for your comma 5 files. But note that if you try to load this script, it will object to **dt-load-object** since this variable only has a value when the comma 5 file is being loaded by the system (when its directory is being opened). See section 3 for important information on how to actually create and load comma 5 files.

Comma 21

The comma 5 file has told the new class of document that it should look in the comma 21 file to find its open method. So, one of the functions we'll want to put into the comma 21 file is the function that serves as the new open method:

18–1 addrbk–open

```
(defun addrbk–open (dt–obj &rest args)
(let (addrbk–text–editor)
  ; open the document
  (apply #'tell–next args)
  ; create new text editor
  (setq addrbk–text–editor (obj–new–class doc–text–editor–class
  "addrbk–ed"))
  (obj–provide addrbk–text–editor mid:get–custom
'addrbk–custom–method)
  (addrbk–keys)  ; redefine some keys
  (addrbk–reclass–buttons)   ; make some graphics active buttons
  (tell *text–editor* mid:set–class addrbk–text–editor)

  ))
```

This open function avoids a mistake that everyone makes at least once. The mistake is to write an open function that adds the additional functionality we want but which then tries to open the document by telling it *(tell dtobj mid:open)*. In that case, the document sees that it has to open itself and that its open method is the function *addrbk–open*. So now it runs the function again, sees that it has to open itself, runs the function again, sees that it has to open itself ... forever.

So, the function above uses *tell–next* which gets the method of the object's parent class. In this case, the parent of the address-book document class is the normal document class. The normal document class's *open* method does the usual things when a document opens. The *tell–next* function should only be used in code that is giving an object a new method (in this case, it's used in the redefinition of the address-book class's *open* method).

Now that the document is open, the address book's *open* method creates a new text editor class and gives that text editor a new custom popup. It goes to a function that builds a modified keymap for the document so that the document's keys will act differently. Finally, it tells the document's text editor to reclass itself so that it can pick up the new characteristics we've just defined.

Simple example

Let's take a simple example. This document has a new open method that prompts the user for which view of the document she wants; depending on her answer, it either turns on or off the local control expression.

Let's begin with the comma 5 file, which will make a new class of document called "two-vers-class":

```
(unless (boundp 'two-vers-class)
(setq two-vers-class
  (obj-new-class dt-document-class "Two Version Doc"))
  (defautoload two-vers-do-open
(dt-path *dt-load-object* :part 21))
)
  (obj-provide two-vers-class mid:open 'two-vers-do-open)

  (tell *dt-load-object* mid:set-class two-vers-class)
```

Now for the comma 21 file, which really is about all there is for this simple document:

18-2 two-vers-do-open

```
(defun two-vers-do-open (dtobj)
(let (show)
  ; do the parent class's open
  (apply #'tell-next args)
  ; ask to turn the local control expression on or off
  (setq show (stk-open "Turn local control expression on?" :yes-no))
  ; turn the expression on or off
  (tell (doc-current-icon) mid:set-props
    :local-control-expression-enabled show)
))
```

Working with active documents

It's a good idea to segregate active documents as much as possible while you're developing them to cut down on the damage inevitable programming errors may cause. At a minimum, as you work on an active document, put it into its own folder.

Because comma 5 and comma 21 are invisible to the desktop, you may find it useful to create linked Lisp files which are visible and which will allow you to do some editing of these files on your desktop. To do this:

- Create a normal Lisp file in the directory where your active document is
- Select the Lisp file and Copy ⧫Link ⧫to original object. Paste the linked file into the directory
- Props the copy and change the pathname so that it points to the comma 5 or comma 21 file

The trickiest part of working with active documents is forcing the system to re-read a comma 5 file which it has read once already. Remember the beginning of our original comma 5 file?

```
(unless (boundp 'addr-bk-class)
  (setq addr-bk-class
    (obj-new-class dt-document-class "Address Bk"))

  ; tell it to look in its comma 21 file to find its new open method
  (defautoload addrbk-open
  (dt-path *dt-load-object* :part 21))
)
```

The first line of this file basically tells the script to run what follows once and only once. As you work on your comma 5 file, you may not want it to keep skipping those initial lines of code. So, change the first line to:

```
(unless nil ; (boundp 'addr-bk-class)
```

This comments out the code and forces the *unless* statement to always be fulfilled.

But we're not out of the woods yet. The system only reads the comma 5 file the first time it opens the directory. And if you try to *load* it using the *load* script in your Select cabinet, the system will tell you it can't successfully evaluate it because it can't find *dt-load-object* since that's an object that's only found when the document itself is having its comma 5 file loaded; the comma 5 file by itself doesn't know what document it's associated with. The following script will force a document's comma 5 file to load:

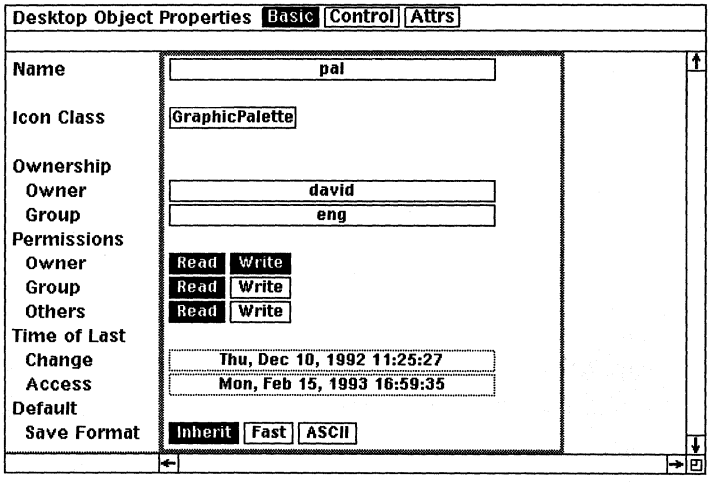
18-3 load-5

```
; make sure we're looking in the current container
(dt-set-container (dt-pointer-container))
; load the comma 5 file
(dt-load-methods (dt-child-selected))
```

Put this into your Selection cabinet. Or, bind it to a key, such as ^Xl:

```
(kbd-bind kbd-dt-map "\^Xl" '(progn
  (dt-set-container (dt-pointer-container))
  (dt-load-methods (dt-child-selected))))
```

Occasionally, after you've succeeded in reclassing a document you will wish you could just open it like a regular document so you can do some document work in it. You can, and no Lisp is required. Just go to the document's prop sheet and change its icon class.



Icon class set to GraphicPalette.

Another example — Forms fill

Here is an example. It is the beginning of what could turn into an automated form:

- While you are in a field in this form, you get a popup that lists the appropriate choices.
- If you enter values into a field, it will check to make sure they are within the appropriate range.
- You can either get "short help" (a stickup) or "long help" (hyperlink to a help document) for any field.
- Hitting the return key while in a field takes you to the next field.

In this example, none of the above functions are implemented with all the error-checking one would want in a real application. Nevertheless, it may give you an idea of how to proceed.

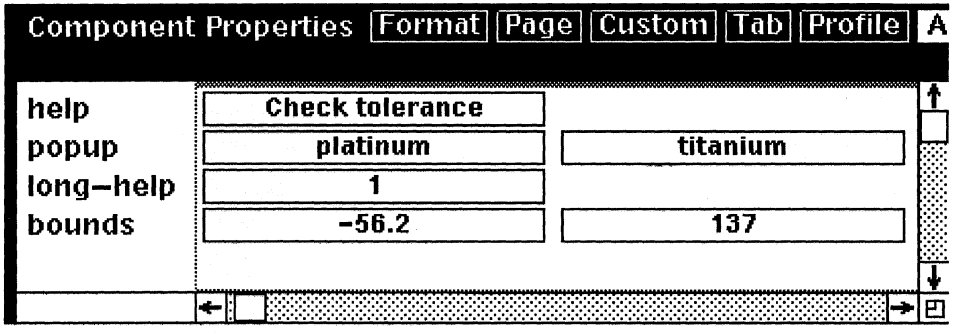
To create this active document, you should create this comma 5 file:

18–4 autoform–comma–5

```
(unless (boundp 'autoform-class)
  (setq autoform-class
  (obj-new-class dt-document-class "Auto Form"))
```

```
  (defautoload autoform-open
  (dt-path *dt-load-object* :part 21))
)

(obj-provide autoform-class mid:open 'autoform-open)
(tell *dt-load-object* mid:set-class autoform-class)
```

The rest of this should go into your comma 21 file. (The description of what it does is in the commented section immediately below.)



Property sheet shows attributes set for guided form active document.

```
;; F1 gets short help
;; F2 gets long help (looks for "longhelp.doc" on the desktop)
;; F3 gets a context-senstive popup for filling in the form
;;
;; Expects form doc to have following attributes:
;;   bounds (low and high): for range checking
;;   popups: determines local popup. Give list.
;;   help: short help text
;;   long-help: page number in longhelp.doc
;;
;; Fields with popups, help, etc. should be inlines named "blank"
;; The attributes described above apply to the inline cmpns, not to
;; the top level ones (although you could put in help there as well)

;; pathname to longhelp document. Change this if you want your long help
;; doc to be elsewhere or otherwise named. (Consider making this an
```

```
;; attribute of the document so you don't have to look at Lisp to change it.)
(setq *long-help-doc* "longhelp.doc")
```

18–5 autoform–short–help

```
(defun autoform-short-help ()
; creates stickup with short help text based on contents of "help" attribute
(let (text)
  ; get the text from the cmpn
  (setq text (tell (doc-point-cmpn) mid:get-attrs "help"))
  (if text
    (stk-open text)
  ; else
  (stk-open "No help available."))
))
```

18–6 autoform–long–help

```
(defun autoform-long-help ()
; goes to right page in longhelp.doc
 (let (page-number)
  ; is there any such file? If not:
  (if (not (probe-file *long-help-doc*))
    (progn
      (stk-open (format nil "Help file ~A not found." *long-help-doc*))
      (quit))
    ; Else, yes, the help doc exists:
    (progn
      ; get the page number from the cmpns long-help attribute
      (setq page-number (tell (doc-point-cmpn) mid:get-attrs
      "long-help"))
      ; tell help doc to open to the page number in the attribute
      (tell (dt-object *long-help-doc*) mid:open :page (atoi
      page-number)))
    )
))
```

18–7 autoform–get–attrs

```
(defun autoform-get-attrs (attribute cmpn)
; get a multi-value set of attributes
(let (attrs values)
  ; get the attributes
  (setq attrs (tell cmpn mid:get-props :attributes))
  ; find the desired attributes in the list that gets returned
  (if attrs (setq values (cdr (assoc attribute attrs 'equal))))
values))
```

18–8 autoform–verify

```
(defun autoform–verify ()
 ; checks to see if we're in an inline named "blank"
 ; and verifies contents against attributes (bounds)
 (let (min max text bounds m m1 (ok t))
 ; are we in a cmpn named "blank"? If not:
 (if (not (string= "blank" (tell (doc–point–cmpn) mid:get–name)))
   (progn
     ; go to next line
     (autoform–goto–next–line (doc–point–top–cmpn))
     (quit))) ; quit
 ; in proper inline
 ; get min and max values from attributes
 (setq bounds (autoform–get–attrs "bounds" (doc–point–cmpn)))
 ; if not any bound listed, then quit
  (if (not bounds) (quit)
 ; get the two values from the list of two that's returned
 (setq min (first bounds))
 (setq max (second bounds))
 ; convert to integer from ascii
 (if min (setq min (atoi min)))
 (if max (setq max (atoi max)))
 ; if no useable min and max bounds, then quite
 (if (or (not min) (not max))
    (quit))
 ; there are min and max
 ; get value of text that was entered
 (setq m (tell (doc–point–cmpn) mid:get–marker))
 (setq m1 (tell (doc–point–cmpn) mid:get–marker t))
 (setq text (tell m mid:get–substring m1 t t))
 (if text
   (progn
   (setq text (atoi text))
   (if text
     (progn
       ; do the compare of min and max
       (if (or (not (>= text min))
               (not (<= text max)))
         ; out of bounds
       (progn
         (setq ok nil)
         ; create stickup announcing out of bounds
```

```
        (stk-open
          (format nil "Out of range\n~D-~D" min max)))
    ))))))
      ; if not out of bounds, then go to next line
      (if ok
        (progn
          (autoform-goto-next-line (doc-point-top-cmpn))))
))
```

18–9 autoform–goto–next–line

```
(defun autoform-goto-next-line (c)
 ; finds next inline named "blank". Assumes only one per cmpn!
(let (next-inline inline done)
  (while (not done)
    ; get next cmpn
    (setq c (tell c mid:get-next))
    ; if no more cmpns, go to first blank inline
    (if (not c)
      (progn
        (autoform-goto-first-inline)
        (setq c (doc-point-top-cmpn))))
    ; look for an inline
    (setq inline (tell c mid:get-child))
    ; if you find one, and it's named "blank"
    (if (and inline
      (string= "blank" (tell inline mid:get-name)))
      (progn
        ; flag that we're done looking
        (setq done t)
        (setq next-inline inline)))
    ; if no more cmpns, then stop looking
    (if (not c) (setq done t))
    )
    ; if we found the next inline, then go to it
    (if next-inline
      (doc-goto-marker (tell inline mid:get-marker)))
))
```

18–10 autoform–goto–first–inline

```
(defun autoform-goto-first-inline ()
; goes to first inline named "blank" in doc
 (let (c inline done)
  ; get first cmpn
```

```
  (setq c (tell *document* mid:get-child))
  ; loop through all cmpns
  (catch 'done
  (while (not done)
    ; look for an inline in the cmpn
    (setq inline (tell c mid:get-child))
    ; yes, we found an inline and it's named "blank"
    (if (and inline
         (string= "blank" (tell inline mid:get-name)))
      ; so flag that we're done
      (progn
      (setq done t)
      (throw 'done))
      ; otherwise, no inline named "blank," so get next cmpn
      ; else
      (setq  c (tell c mid:get-next)))))
  ; if we found one, then go to it
  (if done (doc-goto-marker (tell inline mid:get-marker)))
))
```

18–11 autoform–insert

```
(defun autoform-insert (&optional data)
; inserts into cmpn the text from the popup
(let (m m1 (c (doc-point-cmpn)))
  ; delete current contents
  (setq m (tell c mid:get-marker))
  (setq m1 (tell c mid:get-marker t))
  (tell m mid:delete m1)
  ; insert text
  (tell m mid:insert (car data))
))
```

18–12 autoform–popup–method

```
(defun autoform-popup-method (&optional data d dd)
; builds popup out of list in "popup" attribute
(let (autoform-popup attr-vals text (vplist nil))
  ; get attributes
  (setq attr-vals (autoform-get-attrs "popup" (doc-point-cmpn)))

  ; create list of popup entries
  (while (setq text (pop attr-vals))
    (push (popup-create-entry text 'autoform-insert nil text) vplist))
    ; create popup by applying popup-create-list to list of "popup" attr
```

```
    ; values
    (setq autoform-popup (apply #'popup-create-list vplist))

  ; do the popup
  (popup-activate autoform-popup)
))
```

18–13 autoform–alter–keys

```
(defun autoform-alter-keys ()
 ; assign special keys
(let (autoform-map)
  (setq autoform-map (kbd-make-sparse-keymap))
  (kbd-bind autoform-map
    "\^X*F01" 'autoform-short-help    ; F1
    "\^X*F02" 'autoform-long-help ; F2
    "\^M" 'autoform-verify)           ; Return key
  ; install new keys
  (tell *document* mid:set-props :keymap (list autoform-map
kbd-doc-map))
))
```

18–14 autoform–open

```
(defun autoform-open (dt-obj)
(let (autoform-text-editor)
  ; open
  (apply #'tell-next args)
  (wn-message () "Initializing ...")
  ; create new text editor to give it a new popup
  (setq autoform-text-editor (obj-new-class doc-text-editor-class
  "autoform-ed"))
  (obj-provide autoform-text-editor mid:popup
  'autoform-popup-method)
  (tell *text-editor* mid:set-class autoform-text-editor)
  (autoform-alter-keys)
  ; go to first inline
  (autoform-goto-next-line (tell *document* mid:get-child))
  ; new text editor
  (wn-message () "")
))
```

User interface

Now we're going to create a much more complex (and more useful) active document, focusing on some user interface pieces. The active document is a phone directory that sorts entries alphabetically.

Here's the comma 5 file:

```
; if not addrbk class, make one
(unless (boundp 'addr-bk-class)
  (setq addr-bk-class
    (obj-new-class dt-document-class "Address Bk"))
; look for open method in comma 21 file
(defautoload addrbk-open
    (dt-path *dt-load-object* :part 21))
)
```

The comma 5 file causes the system to look in the comma 21 file for the function that is *addr-bk-class's* new open method. Here is the relevant open function:

```
(defun addrbk-open (obj)
  ; open the addressbk
  (apply #'tell-next obj)
  ; create new text editor
  (setq addrbk-text-editor (obj-new-class doc-text-editor-class
  "addrbk-ed"))
  ; give new text editor a custom method to hang popup off of it
  (obj-provide addrbk-text-editor mid:get-custom
  'addrbk-custom-method)
  ; make this doc's text editor an instance of the new one
  (tell *text-editor* mid:set-class addrbk-text-editor)
  ; bind some keys for this doc
  (addrbk-keys)
  ; turn some graphics into buttons
  (addrbk-reclass-buttons)
)
```

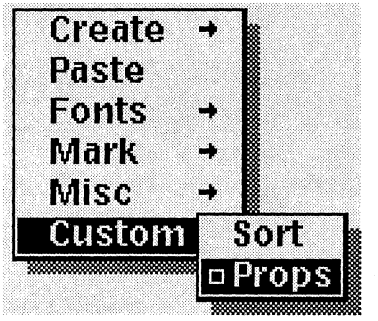
We want this document's text editor to have its own popup, which we'll hang off of the *Custom* entry. We laid the groundwork for this in the *open* method by giving the text editor a new custom method (which it will hang off the bottom of the text popup). Now let's define the popup method:

18–15 addrbk-custom-method

```
(defun addrbk-custom-method (&optional a b)
  addrbk-custom-popup
)
```

Now we'll build the actual popup.

```
(setq addrbk-custom-popup
  (popup-new-list
    (list
      (popup-new-entry "Sort" :handler 'sort-addrbk)
      (popup-new-entry "Props" :handler 'addrbk-do-propsheet))))
```



Custom popup for address book.

Later we'll define the *sort-addrbk* and *addrbk-do-propsheet* functions referenced here. But now let's continue with the user interface by redefining what the *Escape-Pagedown* key does within this document; we'll rebind it so that it takes us to the next letter (i.e., take you to the B's, if you're in the A's) , rather than to the next page. The following function is called by the *open* method.

18-16 addrbk-keys

```
(defun addrbk-keys ()
 ; assign the keys
(let (addrbk-map)
  ; make a blank keymap
  (setq addrbk-map (kbd-make-sparse-keymap))
  ; on that map, bind the ESC-Enter key so that it invokes a new function
  (kbd-bind addrbk-map "\^[\^X*Next" 'addrbk-goto-next-letter)
  ; tell this document that its keymap should consist of the new one and the
  ; usual one
  (tell *document* mid:set-props :keymap (list addrbk-map
kbd-doc-map))
))
```

Now let's write the function we just attached to a key.

18-17 addrbk-goto-next-letter

```
(defun addrbk-goto-next-letter ()
```

```
; goto next letter divider cmpn
(tell *cmpn-editor* mid:goto :next "letter"))
```

Now we want to create a graphic that will act as a button, so that when it's clicked, something will happen. To do this, we're going to make a new class of graphic that when selected *eval's* whatever may be the value of an attribute called *do*. We'll begin by making the new class:

```
; use "unless" so we only do this if we haven't already created this new
; class
(unless (boundp 'addrbk-button-class)
  (progn
    ; create the new class
    (setq addrbk-button-class (obj-new-class dg-named-class
    "addrbk-button-class"))
    ; give class new selection method
    (obj-provide addrbk-button-class mid:allow-selection
    #'addrbk-select-fcn)))
```

We've given our buttons a new select function, so that when one's selected, it will not just blink but will do something. In this case, we want it to *eval* the value of its "do" attribute (if any):

18–18 addrbk-select-fcn

```
(defun addrbk-select-fcn (obj)
  (doc-eval-attribute obj "do"))
```

Now we need to find all the buttons in the document and tell them to take on this new class. We'll take any graphic named "button" as a button.

18–19 addrbk-reclass-buttons

```
(defun addrbk-reclass-buttons ()
; reclasses all diagramming objects named "button" to
; addrbk-button-class
(let (pool head inst)
  (when ; traverse name pool of named diagramming objects
    (setq pool (name-find-pool dg-named-class))
    (setq head (tell pool mid:get-child))
      (while head  ; find all instances named "button"
        (setq inst (tell head mid:get-child :along :name))
          (while (and inst (string= (tell inst mid:get-name) "button"))
            (tell inst mid:set-class addrbk-button-class)
              (setq inst (tell inst mid:get-next :along :name)) )
              (setq head (tell head mid:get-next))
```

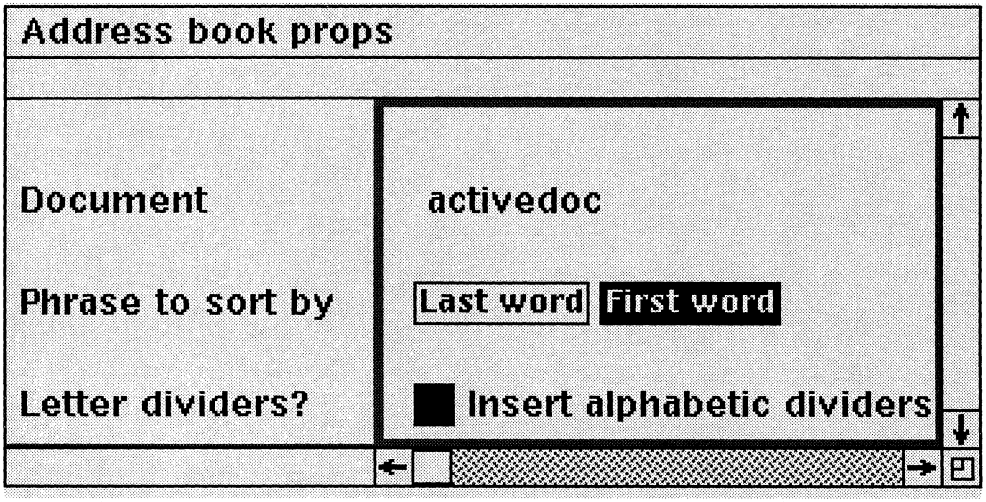
```
  ) ; while head
 )))
```

This function goes to the name pool, looks at all named graphic objects, finds the name "button," and visits every graphic named button, telling it is now reclassed as an *addrbk–button–class* object.

The last thing we need to do is go to into our document and create a graphic, name it "button," create an attribute called "do," and give the graphic's "do" attribute the value of "(sort–addrbk)," which is the function we want activated when the user presses the button. Then we want to do a *Props* on the button's frame, turning off "Frame Selection" and "Border Visible" so that the user can click directly on the graphic itself. Also, print lock it so that when you print out your address book, you won't have to see it. We also want to make the frame a repeating shared contents frame so that you can have the button on every page.

Finally, we want to create a property sheet. (This is what the user will see when she selects "Props" from the Custom popup menu we designed earlier.)

First, let's design the look and functionality of the prop sheet.



Property sheet for configuring address book.

18–20 addrbk–submenu–open

```
(defun addrbk–submenu–open (sheet)
  ; design prop sheet
  ; start
  (tell sheet mid:start 18)
  ; insert blank line
  (tell sheet mid:line "")
```

```
  ; put in name of document affected
  (tell sheet mid:line " Document")
  (tell sheet mid:text (tell *document* mid:get-name) 2)

  ; radio button
  (tell sheet mid:line "")
  (tell sheet mid:line " Phrase to sort by")
  (tell sheet mid:button (list "Last word" "First word")
    2 0 'sort-key-fcn :type :list :initial (addrbk-get-data :sort-by 1))

  (tell sheet mid:line "")
  ; put in letter dividers?
  (tell sheet mid:line " Letter dividers?")
  (tell sheet mid:button " " 2 2 'divider-fcn :type :toggle :initial
(addrbk-get-data :divider 0))
  (tell sheet mid:text "Insert alphabetic dividers" 4)

  ; end
  (tell sheet mid:end)
)
```

Now we'll set up some defaults for when the prop sheet first opens:

```
; set default props for when the prop sheet first appears
; sort by first words (i.e., assume "Smith, Jane" format)
(setq  *addrbk-sort-by  1
       *addrbk-divider*  t      ; put in alphabetical dividers
)
```

Our property sheet has some buttons that call functions. Now we have to define the functions.

18–21 sort-key-fcn

```
(defun sort-key-fcn (old new)
; sets global to tell us whether to sort on first or last name
  (setq *addrbk-sort-by* new)
t
)
```

18–22 divider-fcn

```
(defun divider-fcn (old new)
; sets global (integer) to tell us whether to insert letter dividers
  (setq *addrbk-divider* new)
t)
```

Now let's create a popup for the property sheet. It will have a "Close" and an "Apply" choice.

18–23 addrbk–popup

```
(defun addrbk–popup (menu window)
; create prop sheet popup
  (popup–activate
    (popup–create–list
      (popup–create–entry "Apply" 'addrbk–apply–fcn nil window)
      (popup–create–entry "Close" 'addrbk–close–prop–sheet nil
window)
)))
```

The next two functions are what the prop sheet popups call. The first closes the property sheet. The second applies the changes by saving the data.

18–24 addrbk–close–prop–sheet

```
(defun addrbk–close–prop–sheet (window)
; close method for popup window
  ; make the window nil
  (setq addrbk–window nil)
  ; do the close
  (tell (car window) mid:close)
)
```

18–25 addrbk–apply–fcn

```
(defun addrbk–apply–fcn (&optional data)
  ; apply the prop sheet data
  (tell *document* mid:put–saved–data  :sort–by *addrbk–sort–by*)
  (tell *document* mid:put–saved–data  :divider *addrbk–divider*)
)
```

When the active document does a sort, it will look to the global variables saved in the apply function to see what type of sort it ought to be.

Now we need the code that actually creates the prop sheet:

18–26 addrbk–do–propsheet

```
(defun addrbk–do–propsheet (&optional data)
(let (addrbk–submenu addrbk–menu addrbk–window)
  ; create a new prop sheet object
  (setq addrbk–submenu (tell–class prop–submenu–class mid:new))

  ; give it a method for opening a submenu
```

```
  (tell addrbk-submenu mid:provide mid:open 'addrbk-submenu-open)

  ; create prop sheet obj
  (setq addrbk-menu (tell-class prop-menu-class mid:new))

  ; give prop sheet a popup
  (tell addrbk-menu mid:provide mid:popup 'addrbk-popup)

  ; create a submenu
  (tell addrbk-menu mid:set-props :submenus (list addrbk-submenu))

  ; open the prop sheet
  (setq addrbk-window (tell addrbk-menu mid:open "Address book
props"
    (car (tell *wn-wmgr* mid:get-props :pointer-position))
    (cdr (tell *wn-wmgr* mid:get-props :pointer-position))
  ))
))
```

New functions

Now we have to do the work of writing the functions we want this active document to have.

First we'll get the saved data that tells us what the user's preferences are, based upon her previous interaction with property sheet (or the defaults, if she's never gone to the prop sheet). The following function's first argument is the name of the type of data we're getting (*:sort-by* or *:divider*); the second argument is the type of data we'll be getting.

18–27 addrbk-get-data

```
(defun addrbk-get-data (fun type)
; gets doc's saved lisp data describing user's preferences from last
; interaction with prop sheet
; type is bool, int, string. If no saved data, returns something reasonable
(let (r)
  ; get the saved data
  (setq r (tell *document* mid:get-saved-data fun))
  (if (and (not r)
      (> type 0))
    (progn
      (if (= 1 type)
      (setq r 0)
      ; else
      (setq r "0"))))
```

```
r
))
```

Next is a straightforward function that takes a component and returns all of its text (but only its text).

18–28 addrbk–get–text

```
(defun addrbk–get–text (c)
  (tell (tell c mid:get–marker) mid:get–substring
  (tell c mid:get–marker t) t t)
)
```

Now let's write the sort function. It will find all thes component named "record" and will alphabetize them. (Note: This function expects that you have entered addresses in your address book using a component named "record." So, make sure your address book has a component with that name.)

18–29 sort–addrbk

```
(defun sort–addrbk (&optional arg)
(let (c new–c first–record–marker item (clist nil))
  ; let us back out
  (if (not (stk–open "This will sort all contents. Ok?" :yes–no))
    (quit))
  ; build list of all cmpns named "record"
  (setq c (tell *document* mid:get–child))
  (while c
    (if (string= "record" (tell c mid:get–name))
      (push c clist))
    (setq c (tell c mid:get–next)))
  ; sort the list, using a specially–designed sorting test
  (setq clist (sort clist 'addrbk–sort–test))
  ; delete all records and letter dividers
  (tell *cmpn–editor* mid:select "record")
  (tell *cmpn–editor* mid:select "letter")
  (tell *cmpn–editor* mid:cut)
  ; go to the end of the document
  (if (setq first–record–marker
    ( tell (tell–class doc–cmpn–class mid:get–last) mid:get–marker))
      (doc–goto–marker first–record–marker))
  ; insert the new records
  (tell *cmpn–editor* mid:set–props :caret–direction :next)
  (while (setq item (pop clist))
    (setq new–c (tell *cmpn–editor* mid:create "record"))
    (tell (tell new–c mid:get–marker t) mid:insert (addrbk–get–text item))
```

```
  )
  ; insert letters
  (if *addrbk-divider* (addrbk-insert-letters first-record-marker))
  (tell *cmpn-editor* mid:deselect :all)
  (doc-flush-queue)
))
```

Now we need to write the sort test referenced above. This function gets passed two components. It gets the text for each, and then goes to *addrbk-get-last-name* to find what it considers to be the name to sort on.

18–30 addrbk-sort-test

```
(defun addrbk-sort-test (a b)
(let (texta textb)
  (setq texta (addrbk-get-last-name (addrbk-get-text a)))
  (setq textb (addrbk-get-last-name (addrbk-get-text b)))
  (> (string-compare textb texta) 0)
))
```

The above function references *addrbk-get-last-name* that takes the text of a component and returns the last name (which may be the first or last word on a line).

18–31 addrbk-get-last-name

```
(defun addrbk-get-last-name (text)
(let (pos line start-pos end-pos name)
  ; get first line
  (setq pos (string-contained "\n" text))
  (if pos
    (setq text (substring text 0 pos)))

  ; if sorting by first name in line
  (if (equal *addrbk-sort-by* 1)
    (progn
      ; first trim any leading spaces
      (setq text (string-left-trim " " text))
      ; set beginning spot at beginning of line
      (setq start-pos 0)
      ; find the first space, if any
      (setq end-pos (string-contained " " text))
      ; if no space, then there's only one word on the line
      (if (not end-pos)
        (setq end-pos (string-length text)))
    )
    ; else, we're looking for last word on line
```

```
    (progn
      ; first trim any trailing spaces
      (setq text (string-right-trim " " text))
      ; set end position to end of line
      (setq end-pos (string-length text))
      ; get last space on line
      (setq start-pos (string-contained " " text -1 :reverse)))
      ; if there is a space, then advance counter one past it
      (if start-pos
        (setq start-pos (1+ start-pos))
        ; if no space, then set it to 0
        (setq start-pos 0))
  )
  ; get the text
  (substring text start-pos end-pos)
))
```

The next function we want to add is one that automatically inserts letter dividers. We only want to insert the letters for which there are addresses with instances. This function gets passed a marker in the component immediately before the *record* components begin.

18–32 addrbk-insert-letters

```
(defun addrbk-insert-letters (m)
; insert the distinguishing letters. M is marker in first cmpn before records
(let (c cur-letter letter new-c mark)
  ; get parent component of the marker
  (setq c (tell m mid:get-parent))
  ; get the next component, i.e., where the addresses begin
  (setq c (tell c mid:get-next))
  ; set the current letter to an empty string
  (setq cur-letter "")
  ; go through all the remaining components, inserting letter dividers
    (while c
      ; get the first letter of the last name for that component
      (setq letter (string-uppercase
        (string (char
          (addrbk-get-last-name (addrbk-get-text c)) 0))))
      ; is the first letter of this last name different from the one before it?
      (if (not (string= cur-letter letter))
        ; if it's different ...
        (progn
          ; go to the cmpn, and insert the letter cmpn
          (doc-goto-marker (tell c mid:get-marker))
```

```
        (tell *cmpn-editor* mid:set-props :caret-direction :previous)
        (setq new-c (tell *cmpn-editor* mid:create "letter"))
        (setq mark (tell new-c mid:get-marker t))
        ; insert the first letter into the letter cmpn
        (doc-goto-marker mark)
        (tell mark mid:insert letter)
        ; update the current letter variable
        (setq cur-letter letter)
    ))
  ; get the next cmpn
  (setq c (tell c mid:get-next))
  ) ; end of while
))
```

Chapter 19
Fifteen Errors You Will Make

Some errors are so easy to make in Lisp that you are practically predestined to make them. Here are some of my favorites, in no particular order.

But first, remember the keys to breaking out of an infinite loop. Control–Z will give you the first level break. If that's not sufficient, hit ^Z again and choose "Stop Lisp." If that doesn't work, try unplugging your machine. If that doesn't work, you've got the basics for a bad science fiction movie. (DOS users note: Control–Break should give you the "Stop Lisp" stickup; ^Z^Z won't!)

Statement doesn't eval

```
(if sky-is-gray
  (take-umbrella t)
  (wear-galoshes t))
```

If an *if* statement is true, only the next statement is *eval*'ed. The one right after it is treated as an *else*. In this example, if *sky–is–gray* is true, then *take–umbrella* will be *eval*'ed, but *wear–galoshes* won't be unless *sky–is–gray* is nil.

So, if you really wanted to *take–umbrella* and *wear–galoshes* if the *sky–is–gray*, you should have written:

```
(if sky-is-gray
  (progn
  (take-umbrella)
  (wear-galoshes)))
```

The values at a breakpoint make no sense

You've put in a breakpoint, using the *(break)* function so you can examine the values at a particular point in the function. But the symbols you want to examine don't even exist within that function, or their values are screwy.

You probably have a breakpoint left in from a previous debugging session. Use your text editor to search for it and get rid of it.

Changes to text formatting don't take effect

Your script makes some series of changes to a document, but they don't occur. What's worse, if you enter the same commands by hand (through the *Listener* or *ReadEval* stickup, or some other way of doing them one at a time), they work fine. So you know your code is ok.

There's a good chance that you've run into a problem that occurs when you try to affect a document with a series of commands without giving the document time to catch up with you.

To avoid this, insert *(doc–flush–queue)* after each document change statement. This causes the document to apply any pending changes so that the next change you want to apply is operating on the properly adjusted document. The problem with this, however, is that adjusting the document after each command takes longer than letting the system "queue" the changes, In part this is because *doc–flush–queue* causes the document to redisplay itself. If you are walking through a complex table, you really don't want the entire table to redisplay after each change you apply. A way to avoid this is to ... display stuff. Sometimes *(doc–flush–queue)* isn't enough. The next thing to try is *(event–after–put 'function)*. For example, you could apply italics by the line *(event–after–put '(tell *text–editor* mid:set–props :italic t))*.

String error

Your script that manipulates strings keeps ending in error messages.

Quite possibly you're feeding some string functions nil values. They don't like that. For example:

```
(string–contained "a" my–string)
```

will generate an error if my–string is nil. So, do the following:

```
(if my–string
  (string–contained "a" my–string))
```

Changes to a comma 5 file don't take effect

Typically in a comma 5 file, you test to see if the new class of document has already been declared so that you won't declare it every time the comma 5 file gets *eval*'ed.

```
(unless  (boundp 'dt-hpers-class)
  (setq dt-hpers-class
  (obj-new-class dt-document-class "HangPerson")
  )

  (defautoload hpers-open
    (dt-path *dt-load-object* :part 21))
)

(obj-provide dt-hpers-class mid:open 'hpers-open)
```

But that means that while you're debugging, some portion of the comma 5 is being skipped. If you don't want to skip it, do the following, remembering to change it back when you're done:

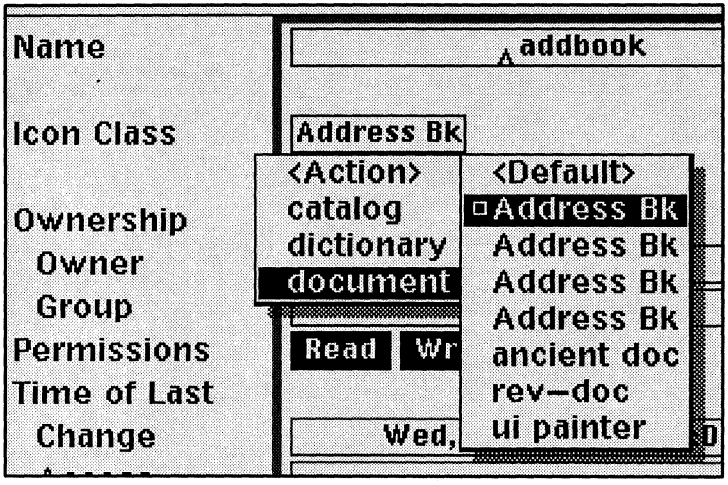
```
(unless nil ; (boundp 'dt-hpers-class) <======
  (setq dt-hpers-class
  (obj-new-class dt-document-class "HangPerson")
  )

  (defautoload hpers-open
    (dt-path *dt-load-object* :part 21))
)

(obj-provide dt-hpers-class mid:open 'hpers-open)
```

Your active document is spawning too many classes

On the document property Icon Class popup, you get more than one listing of some class of document you've created. Reread the previous error. Your comma 5 file probably isn't testing to see if the class exists before redeclaring it.

Changes to comma 21 file have no effect

You've made some changes to the comma 21 file that defines some functions for an active document you've built. But the changes just don't seem to be taking effect. You've even stuck in a *(stk-open "In 21")* at the beginning of the file, but it doesn't get *eval*'ed either.



Whoops.

Make sure that you've saved the file before opening the document. The comma 21 file that will get loaded when the document opens is the currently saved one.

Your code continues to make mistakes you corrected quite a while ago

If you're working in a comma 21 file, make sure there isn't an active document of the same type that is getting loaded after your most current version, overwriting the functions in the new one.

In some early versions, a Lisp script or a comma file that gets cut (and therefore put onto the clipboard) gets *eval*'ed while actually on the clipboard. Purge it from the clipboard.

Buttons on your stickups don't work as predicted

```
(setq choice
  (stk-open "This is a sample stickup with two buttons"
   :buttons (list "Button A" "Button B")))
```

If this is what your code looks like, remember that stickups are zero-based, so that if the user chooses "Button A," that will set choice equal to zero.

Selected icon isn't found by script

You can see the icon selected as plain as the nose on your face, but your script can't find it.

Remember that the Interleaf desktop allows you to have many directory windows open with many items selected at each. To find the selected icon, you have to tell the system which container to look in. This is handled automatically for any script put into the Selection cabinet and chosen through the Custom menu. Other than that, however, you need to tell the system which container the icon is in. If you are developing a script that will end up in the Selection cabinet, but now isn't finding the right icon, you can force a particular container to become the current one (i.e., the one the system looks in when you do a *(dt-child-selected)*) by putting your mouse cursor into the container and running any script off the Custom menu. You might consider building two "no-op" scripts (that do absolutely nothing) and putting one into your Selection and one into your No Selection cabinets. The script can look like this:

```
(list nil)
```

This does absolutely nothing. But, by running it, you will set the container with the cursor in it as the current container.

Division doesn't work

```
(/ 8 3)
```

returns 2. That's because you are dividing an integer by an integer. To avoid this, make either one or other number a float, i.e., a number with an explicit decimal point:

```
(/ 8 3.0)
```

A string never tests as empty

You're testing a string (called, say, "my-string") to see if it's empty. But it never is.

If the code looks like this:

```
(if (not my-string)
  (do-something))
```

then perhaps you want it to look like this:

```
(if (string= "" my-string)
   (do-something))
```

The point is not to confuse nils with empty strings. They're not the same.

The component editor just cut a whole bunch more than you wanted

If you use the component editor to operate on components, remember that it will make changes to any and all selected components. For example:

```
; select current component
(tell *cmpn-editor* mid:select (doc-point-cmpn))
; cut it
(tell *cmpn-editor* mid:cut)
```

If there were any other components already selected, whether by the program or by the user, they will be cut also. To avoid this, first insert the line

```
(tell *cmpn-editor* mid:deselect :all)
```

Can't push object onto list

You have declared a variable called my-list. Now you want to push an object into it:

```
(let (my-list)
  (push (doc-point-cmpn) list)
)
```

You'll get an error message. To avoid it, first explicitly declare my-list to be nil:

```
(let (my-list)
  (setq my-list nil)
  (push (doc-point-cmpn) list)
)
```

or, in compressed form:

```
(let ((my-list nil))
  (push (doc-point-cmpn) list)
)
```

When you open an active document, you get stuck in an infinite loop that never gets past the open function

If you give an active document a new open method — which is a very common thing to do — you can't then use that open method to open it, or you will get stuck in a loop

forever. You have to be able to get to its old open method. Check the chapter on active documents for how to do this. But the basic idea is this:

- Give the active document a new open method. In that new open method, capture the open method of the active document's parent. Typically, the parent is the doc–document–class.
- Open the document with the parent's open method, through *tell–next*. This will do the expected actions of creating a document window, formatting and composing the document, and displaying it.
- Do the rest of what you want your new open method to do.

If you decide to delay doing the parent's open method until after you have done your special open–method activities, just remember that any of those activities that require the document to be open will fail.

Those are some of the most common errors. But part of the great adventure of Interleaf Lisp is coming up with your own, unique errors, mistakes and impracticalities.

Good luck and happy adventuring!

Index

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

Z

More OnWord Press Titles

Pro/ENGINEER Books

INSIDE Pro/ENGINEER
Book $49.95 Includes Disk

The Pro/ENGINEER Quick Reference
Book $24.95

The Pro/ENGINEER Exercise Book
Book $39.95 Includes Disk

Interleaf Books

INSIDE Interleaf
Book $49.95 Includes Disk

Adventurer's Guide to Interleaf Lisp
Book $49.95 Includes Disk

The Interleaf Exercise Book
Book $39.95 Includes Disk

The Interleaf Quick Reference
Book $24.95

Interleaf Tips and Tricks
Book $49.95 Includes Disk

MicroStation Books

INSIDE MicroStation 5X. 3d edition
Book $34.95 Includes Disk

MicroStation Reference Guide 5.X
Book $18.95

The MicroStation Productivity Book
Book $39.95 Optional Disk $49.95

MicroStation Bible
Book $49.95 Optional Disk $49.95

Adventures in MDL Programming
Book $49.95

Programming With MDL
Book $49.95 Optional Disk $49.95

Programming With User Commands
Book $65.00 Optional Disk $40.00

101 MDL Commands
Book $49.95
Optional Executable Disk $101.00
Optional Source Disks (6) $259.95

101 User Commands
Book $49.95 Optional Disk $101.00

Bill Steinbock's Pocket MDL Programmer's Guide
Book $24.95

MicroStation for AutoCAD Users
Book $29.95 Optional Disk $14.95

MicroStation for AutoCAD Users Tablet Menu
Tablet Menu $99.95

MicroStation 5.X Delta Book
Book $19.95

Managing and Networking MicroStation
Book $29.95 Optional Disk $29.95

The MicroStation Database Book
Book $29.95 Optional Disk $29.95

The MicroStation Rendering Book
Book $34.95 Includes Disk

INSIDE I/RAS B
Book $24.95 Includes Disk

The CLIX Workstation User's Guide
Book $34.95 Includes Disk

Build Cell
Software $69.95

SunSoft Solaris Series

The SunSoft Solaris 2.* User's Guide
Book $29.95 Includes Disk

SunSoft Solaris 2.*for Managers and Administrators
Book $34.95 Optional Disk $29.95

The SunSoft Solaris 2.* Quick Reference
Book $18.95

Five Steps to SunSoft Solaris 2.*
Book $24.95 Includes Disk

One Minute SunSoft Solaris Manager
Book $14.95

SunSoft Solaris 2.* for Windows Users
Book $24.95

The Hewlett Packard HP-UX Series

The HP-UX User's Guide
Book $29.95 Includes Disk

HP-UX For Managers and Administrators
Book $34.95 Optional Disk $29.95

The HP-UX Quick Reference
Book $18.95

Five Steps to HP-UX
Book $24.95 Includes Disk

One Minute HP-UX Manager
Book $14.95

HP-UX for Windows Users
Book $24.95

CAD Management

One Minute CAD Manager
Book $14.95

Manager's Guide to Computer-Aided Engineering
Book $49.95

Other CAD

CAD and the Practice of Architecture: ASG Solutions
Book $39.95 Includes Disk

INSIDE CADVANCE
Book $34.95 Includes Disk

Using Drafix Windows CAD
Book $34.95 Includes Disk

Fallingwater in 3D Studio: A Case Study and Tutorial
Book $39.95 Includes Disk

Geographic Information Systems

The GIS Book. 3d edition
Book $39.95

DTP/CAD Clip Art

1001 DTP/CAD Symbols Clip Art Library: Architectural
Book $29.95

DISK FORMATS:
MicroStation
DGN Disk $175.00 Book/Disk $195.00

AutoCAD
DWG Disk $175.00 Book/Disk $195.00

CAD/DTP
DXF Disk $195.00 Book/Disk $225.00

Networking/LANtastic

Fantastic LANtastic
Book $29.95 Includes Disk

The LANtastic Quick Reference
Book $14.95

One Minute Network Manager
Book $14.95

OnWord Press Distribution

End Users/User Groups/Corporate Sales

OnWord Press books are available worldwide to end users, user groups, and corporate accounts from your local bookseller or computer/software dealer, or from HMP Direct: call 1-800-526-BOOK or 505-473-5454; fax 505-471-4424; e-mail to ORDERS@BOOKSTORE.HMP.COM; or write to High Mountain Press Direct, 2530 Camino Entrada, Santa Fe, NM 87505-8435.

Wholesale, Including Overseas Distribution

We have international distributors. Contact us for your local source by calling 1-800-ONWORD or 505-473-5454; fax to 505-471-4424; e-mail to ORDERS @BOOKSTORE.HMP.COM; or write to High Mountain Press/IPG, 2530 Camino Entrada, Santa Fe, NM 87505-8435, USA.

Comments, Corrections, and Bug Fixes

Your comments can help us make better books. If you find an error in our products, or have any other comments, positive or negative, we'd like to know! Please contact our e-mail address: READERS@HMP.COM, or write to us at the address below.

OnWord Press, 2530 Camino Entrada, Santa Fe, NM 87505-8435 USA

Desktop Publishing

ADVENTURER'S GUIDE TO INTERLEAF LISP

Everything the power user needs to begin reprogramming the world's most powerful document tool—Interleaf 5

Interleaf 5 is a publishing powerhouse. Unlocking the potential of Interleaf 5 with Interleaf Lisp can be a true adventure. Take this guide and follow along with lots of sample functions that show you how to customize the user interface, add functionality, and even create "active documents" that respond to your input and automate everyday tasks. Increased productivity is at your fingertips.

TOPICS COVERED INCLUDE:

THE DESKTOP TEXT
POPUPS STICKUPS AND STAYUPS
PROPERTY SHEETS GRAPHICS
TABLES ACTIVE DOCUMENTS

Designed for the hobbyist programmer and Interleaf power user, *Adventurer's Guide to Interleaf Lisp* offers many tips and tricks not found in the manufacturer's reference manuals. Learn to use Interleaf Lisp to maximize productivity in your publishing process.

The bonus disk includes all the Lisp examples shown in the book, plus several useful utilities.

DOS/Windows	UNIX	MAC
YES	YES	NO

Beginner	Intermediate	Advanced	Programming
No	Yes	Yes	Yes

ONWORD PRESS
Dedicated to the Fine Art of Practical User Documentation

ISBN 1-56690-042-5
90000 >
EAN
9 781566 900423