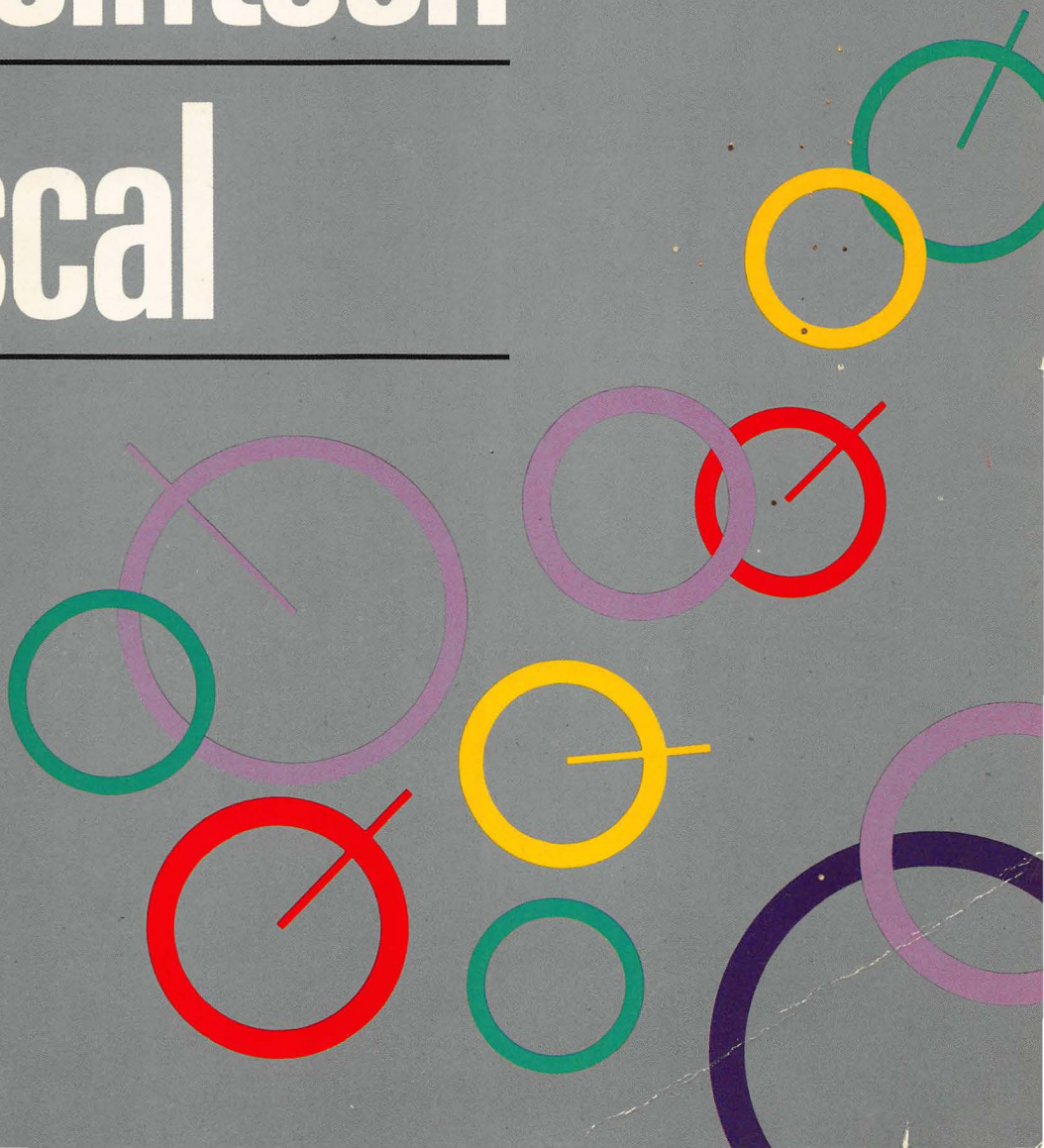
Osborne McGraw-Hill

THE FIRST BOOK OF Macintosh™ Pascal



PAUL A. SAND

THE FIRST BOOK OF MACINTOSH™ PASCAL

THE FIRST BOOK OF MACINTOSH™ PASCAL

Paul A. Sand

Osborne McGraw-Hill
Berkeley, California

Published by
Osborne **McGraw-Hill**
2600 Tenth Street
Berkeley, California 94710
U.S.A.

For information on translations and book
distributors outside of the U.S.A., please write to
Osborne **McGraw-Hill** at the above address.

THE FIRST BOOK OF MACINTOSH™ PASCAL

234567890 DODO 898765

ISBN 0-07-881165-1

Cindy Hudson, Acquisitions Editor
Paul Hoffman, Technical Editor
Catherine Pearsall, Copy Editor
Cheryl Creager, Composition
Yashi Okita, Cover Design

CONTENTS

INTRODUCTION

This book is an introduction to the Pascal programming language and to the version of that language on the Apple Macintosh computer. You can use this book regardless of your level of expertise in computer programming. If you are an absolute beginner, don't worry; it is our assumption that you know nothing about either programming in general or Pascal in particular. We will explain everything you need to know in order to write, enter, and run programs in Pascal on the Macintosh.

Even if you already have some familiarity with the Pascal programming language, you will find this book useful. Macintosh Pascal is like no other version of Pascal you have ever used—just as the Macintosh itself is like no other computer you have used. We will attempt to explain and demonstrate many of the major features of Macintosh Pascal, especially those that aren't found in "Standard" Pascal.

Book Philosophy

Computer programming is a skill that is learned by doing. No book yet written can take the place of the actual experience you will gain in designing and writing your own programs. Realizing this, we will not promise that simply reading this book will make you a seasoned expert in Macintosh Pascal; our goals are more modest. This book will

- Provide you with a description of Macintosh Pascal's features and how these features are used in programs.

- Give you the *rules* of the language: what is legal, what isn't legal.
- Show you many examples of Macintosh Pascal programs that demonstrate how the language can be used to access many of the hardware and software features of the Macintosh itself.
- Provide you with some suggestions for changing the programs presented here as well as ideas for your own programs. Ideally, this should get you started in learning how to program on your own.

In general, the emphasis in this book will be on the final two points: to show you how Macintosh Pascal works and to spark your own curiosity enough to discover more on your own. We will lean heavily on example programs throughout the text to accomplish this.

Somewhat less emphasis will be placed on the more "formal" aspects of Pascal: the precise rules that make one Pascal statement legal and another one unacceptable. Many of the formal rules of Pascal are designed to make sense of very unusual program constructions, ones unlikely to occur in everyday programming. We will forego explaining such rules in all their mind-numbing detail. Instead, we will concentrate on developing your intuitive sense for what's right and what's wrong in Pascal.

As a user of Macintosh Pascal, you own or have access to the documentation provided with the Macintosh Pascal software. You should look upon this book as a *supplement* to the Macintosh Pascal documentation, not as a replacement. The Macintosh Pascal reference manuals are complete, concise, and rigorous. This approach has the advantage that you may look there for the exact rules on what you may or may not do in your Pascal programs. The disadvantage is that it is difficult for programming novices and other programmers not familiar with Macintosh Pascal to sort through the descriptions of every last nit-picking rule and restriction to extract the truly useful information present in the reference manuals: the knowledge you will need *every* time you write a Pascal program.

In this text, we provide a more leisurely and less formal introduction to Macintosh Pascal. We concentrate on the "good parts" of Macintosh Pascal: those that can be used easily by both programmers unfamiliar with either Pascal and

programmers who know Pascal but may not be acquainted with the Macintosh's version of the language.

Why Learn to Program?

As you probably know, everything the Macintosh (or any other computer) does is controlled by its software: programs that tell the computer what to do in various situations and how to accomplish useful tasks. Learning to program is, simply stated, discovering how to tell the computer to "do things."

For many people, computer use is restricted to using programs someone else has written. Given the increasing sophistication and usefulness of commercially available computer software, this is a perfectly workable strategy for many, if not most, computer users. You have an inalienable right *not* to learn to program your computer, if you so choose.

Having said that, however, here are some reasons why you might, after all, want to undertake the task of writing your own programs:

- Learning to program will give you a better idea of how the computer works and what it can and cannot do.
- Programming is, like mathematics, an intellectual discipline that is inherently worth knowing.
- Knowing how to program can stand you in good stead when a computer problem arises at work or at home for which no available prewritten software applies. Depending on your expertise, you may be able to write a program to solve the problem, often with less time and expense than if you had sought commercially available software for the same purpose.
- Programming can be profitable. People who don't know how (or don't have time) to program will pay you money to make their computer jump through designated hoops.
- Last, but not least, programming is fun—a truly endless source of amusement. If you enjoy intellectual challenge, thinking a problem through to its solution, and the sense of accomplishment that occurs when something works as it should, you will find programming to be immensely enjoyable. In a sense, programming is the ultimate computer game.

Why Learn Pascal?

If you decide to learn to program, you need to learn at least one programming language. In this book, obviously, we are suggesting you learn the programming language called *Pascal.* Pascal was developed in the late 1960s by Niklaus Wirth. His aims were to produce a language suitable for teaching programming concepts clearly and systematically and to make the language usable on a large number of computers. His success is obvious: Pascal's popularity has increased rapidly since its introduction, and versions of the language are available on many computers, ranging from small, inexpensive personal computers to large mainframe systems.

Why is Pascal so popular? The primary reason is that Pascal makes the job of writing, reading, and modifying computer programs easier than do many other programming languages.

Here are some ways in which Pascal is a convenient language for programmers. (You need not worry if you don't understand all or any items on this list, by the way. We'll be seeing how all these things work later.)

- Pascal has "structured" control statements (**while**, **repeat**, **for**, **case**, and **if-then-else**) that allow the programmer to write clear and concise code with the flow of control proceeding from top to bottom. The control flow in programs written in languages lacking these structured control statements often contains complex webs of **if** tests and **goto** statements. Such programs are often called "spaghetti code" because they are so difficult for even experienced programmers to untangle.
- Pascal permits the programmer to break up a large program into smaller, relatively independent procedures and functions, each one of which performs a single, easily understood task. Each procedure or function can have its own set of "private" variables that are only used when that procedure or function is executed. Each procedure or function has well-defined input and output parameters used for communicating with its calling routine. This decomposition of a large program into modules greatly aids the programmer in both the initial design of the program and also in any subsequent modifications to the program; the modules can also be reused

in subsequent programs, decreasing the overall programming effort.

- Pascal allows programmers to define their own data types and data structures in addition to those already built into the language. Judicious use of this feature can make a program more compact and easy to understand.
- Pascal allows the use of long identifiers for variables, procedures, and functions. This allows the programmer to use names with mnemonic significance, another aid in understanding a program.

Although Pascal is a good computer language, it is not a perfect one. And it isn't suitable for all applications. Rather than go into Pascal's deficiencies here, however, we'll merely note that a good deal of serious software development is carried out in Pascal. And even if Pascal isn't your programming language of choice, you will find that learning Pascal will make learning and using other programming languages much easier.

Why Learn Macintosh Pascal?

As we will see in the first chapter, the "programming environment" provided by the Macintosh Pascal software is extraordinarily easy to use. In order to write Pascal programs on a typical computer system, you have to learn how to use a number of different programs, each with its own set of hard-to-remember commands and rules. This, fortunately, isn't the case with Macintosh Pascal: there are no complex command sequences to memorize, nor are there separate "editor," "compiler," or "debugger" programs to master. You need only learn to use the Macintosh Pascal software in order to be able to enter, run, and modify your own Pascal programs.

Macintosh Pascal uses most of the conventional elements of the Macintosh user interface: pull-down menus, multiple overlapping windows on the screen, mouse selection, and so on. In fact, if you know how to use a word processing program like MacWrite or Microsoft Word, you already know nearly everything you need in order to enter your Pascal programs into the computer.

Macintosh Pascal is therefore an excellent way to learn

the Pascal language. But that's not all: built into Macintosh Pascal is the capability for you to control nearly all aspects of your computer's operations. These capabilities include some not offered with languages costing many times more than Macintosh Pascal. (As before, you shouldn't worry if you don't understand everything—or anything—on this list as yet.)

- Macintosh Pascal provides a variety of numeric data types that make it possible to write precise and reliable computational programs.
- Macintosh Pascal supplies a number of extensions to the Standard Pascal language that make writing common application programs easier: there is a built-in **string** data type, an **otherwise** clause on the **case** statement, sophisticated memory management, and direct-access file I/O routines, to name a few. (These final two topics are relatively advanced, however, and won't be considered in this text.)
- Macintosh Pascal allows programs to access most of the capabilities of the Macintosh; for example, built-in routines callable from Pascal programs allow your programs to manipulate windows, use the mouse, the clock, and the sound generator.
- Probably the most interesting feature is Macintosh Pascal's ability to call the QuickDraw routines and other software contained in the Macintosh's read-only memory. QuickDraw is an extensive "library" of routines that allows your programs to perform awesome feats of graphics magic.

Macintosh Pascal, while an excellent learning environment, is not suited to writing large application programs. In computer jargon, Macintosh Pascal is an interpreted language, rather than a compiled one. Compared to programs written in other languages, you may observe that your Macintosh Pascal programs are rather slow. (They may very well be fast enough for your purposes, however.) Also, your Pascal programs can't run "by themselves"; the Macintosh Pascal system must be present in the computer's memory at the same time. This imposes a relatively restrictive upper limit on the size of your Pascal programs.

On balance, however, Macintosh Pascal is an excellent learning and programming tool for all but the most demanding applications.

Hardware Requirements

Macintosh Pascal will run on any Apple Macintosh computer; it will also run on an Apple Macintosh XL (Lisa) computer system set up with the MacWorks Macintosh-emulation software. You may run Macintosh Pascal on a bare-minimum Macintosh system with 128K bytes of memory and the single built-in disk drive.

As with most software for the Macintosh, additional hardware will make working with Macintosh Pascal easier. A second disk drive, while not required, will greatly decrease the time you spend on the drudgery of removing and inserting disks. If you add a printer to the system, you'll be able to get listings of your Pascal programs on paper. Expansion to 512K bytes of memory will permit you to write and use much larger programs under Macintosh Pascal than is possible with a 128K system. (Memory expansion will also allow your other Macintosh software to handle large amounts of data more easily.) All programs in this book will fit easily in a 128K Macintosh, however.

How to Read the Rest of the Book

As previously indicated, computer programming is one of those subjects that can't be learned by simply reading a book (even a *good* book). So in order to get the most out of this book, you should read it in front of your computer. Try the example programs in each chapter as you read. Don't be afraid to experiment with the example programs; we'll even suggest some possible experiments to you as we go along.

Even more important to learning a computer language is the ability to take the descriptions and examples of the language elements given here and apply them when the time comes to write programs of your own. This, too, is an ability that only comes with practice. In addition to the experiments you can try out on our example programs, we'll suggest some problems you can try to solve by writing your own programs "from scratch."

If you are a novice to programming, simply start at the beginning. You will find that the material in most chapters depends heavily on information from previous chapters, so

skipping ahead to a seemingly more interesting chapter is generally a bad idea.

If you already know some other version of the Pascal language, you will probably not want to plod through the chapters of this book that contain material you already know. Macintosh Pascal *is* Pascal, after all, and if you know Pascal, you already know an appreciable fraction of the material covered in this book. You might find it more interesting to take a fast track through "Macintosh Pascal-only" parts of the book, skimming or skipping sections that cover material you already know.

We will make it easy for you to do just that. Material in this book that applies only to Macintosh Pascal will usually be confined to individual sections and chapters, not scattered throughout the text. For easy identification, these sections and chapters will all have the word "Macintosh" in their titles.

Chapter One will introduce you to Macintosh Pascal, taking you through a step-by-step example of writing and running a simple program. All readers should probably work through this chapter.

Chapter Two introduces the fundamental concepts of Pascal: constants, variables, integer, real, and Boolean types, assignment statements, expressions, operator precedence, simple input and output operations, and looping control structures. This chapter contains very little Macintosh-specific material.

Chapter Three discusses advanced editing techniques you'll use in Macintosh Pascal: selecting, cutting, pasting, and copying blocks of text, finding and replacing pieces of your program, and general disk housekeeping. Much of this material will be familiar to those acquainted with other Macintosh software, but some Macintosh Pascal-only rules are discussed.

Chapter Four discusses Pascal's decision control structures: **if-then-else**, **case**, and **goto**; it contains very little Macintosh-only material.

Chapter Five covers Macintosh Pascal's marvelous debugging aids: the Observe and Instant windows, program breakpoints, and step-by-step program execution.

Chapter Six explores six additional Macintosh Pascal data types: characters, strings, long integers, and three additional kinds of real numbers: computational, double, and extended.

Of these six data types, only characters are present in Standard Pascal.

Chapter Seven is an introduction to built-in or *library* functions available for use in your Pascal programs. This includes both standard functions built into nearly every version of Pascal, and Macintosh Pascal functions to accomplish tasks such as string manipulation, binary arithmetic, and other special Macintosh operations.

Chapter Eight considers built-in library procedures in Macintosh Pascal, including the first description of QuickDraw routines.

Chapter Nine discusses how to go about programming your own procedures and functions. This is mostly Standard Pascal material.

Chapter Ten introduces the concept of defining your own data types and illustrates the principle using enumerated and subrange types.

Chapter Eleven covers Pascal's "structured" types: arrays, sets, and records, including variant records.

Chapter Twelve discusses many of Macintosh Pascal's predefined types and how they are used to access additional QuickDraw capabilities, as well as other Macintosh features.

Not covered in this text are "advanced" Pascal topics such as recursion, file I/O, pointers, and handles. Our coverage of QuickDraw, while sufficient to allow you to write useful programs that generate sophisticated graphics, doesn't cover many powerful aspects of QuickDraw that are less easy to use. Advanced sound generation using the four-voice synthesizer, event management, direct "in-line" calls to the Macintosh's ROM Toolbox routines, and use of the Standard Apple Numeric Environment (SANE) library are also not discussed in this book.

Recommended Reading

The original definition of the Pascal language was described in *Pascal User Manual and Report* by Kathleen Jensen and Niklaus Wirth (Springer-Verlag, 1978). This book contains a concise but well-written description of the legalities of the Pascal language. It remains a good introduction to Pascal for those who are familiar with another computer language.

Recently both the American National Standards Institute (ANSI) and the International Standards Organization (ISO) have approved Pascal standards that clear up the minor ambiguities contained in the original Jensen and Wirth Pascal. The "Level 0" ISO Standard is equivalent to the ANSI standard. A good, readable description is found in *Standard Pascal User Reference Manual* by Doug Cooper (W.W. Norton, 1983).

In nearly all cases there is no difference between ANSI/ISO Pascal and Jensen and Wirth Pascal. When we use the term "Standard Pascal," we will be referring to either one. In cases where there is a difference, we will explicitly say which interpretation we are using.

GETTING STARTED WITH MACINTOSH PASCAL 1

The first program to write is the same in all languages...

—*B. Kernighan and D. Ritchie*
The C Programming Language
(Prentice-Hall, 1978)

Our goal in this chapter is to acquaint you with most of the basic operations you'll be performing every time you program in Macintosh Pascal. You will learn how to enter programs into the computer and how to correct the inevitable mistakes you will make in the process. The chapter discusses how to run your program once you have typed it in and how to make further modifications to your program. You'll find out how to save programs on disk and how to print them on your printer.

A REVIEW OF MACINTOSH FUNDAMENTALS

This book will assume that you have at least some experience in operating the Macintosh. Since Macintosh Pascal and most Macintosh applications use the same basic operations, if you have used any other Macintosh program, you already know

just about everything you need to know to use Macintosh Pascal.

If you are a complete novice to the Macintosh, we suggest that you take time to explore the features of the Macintosh, either by using the excellent "Guided Tour" casette tapes provided with your computer or by following the more traditional method of reading the Macintosh user manual.

In order to use Macintosh Pascal, you should be acquainted with the following topics and terms:

- *Moving the mouse.* By moving the mouse around your desktop, you move the *pointer* around the Macintosh screen. Usually the pointer is an arrow pointing north by northwest, but its shape changes depending on where it is pointing and what the Macintosh is doing at the time. (When necessary, the text will refer to the pointer's shape, for example the I-beam pointer.)
- *Clicking.* Many operations involve moving the pointer to a certain object and then pressing and quickly releasing the mouse button. In Macintosh jargon, this is called *clicking* the object. For example, to "click the Pascal disk icon" means to move the pointer to the picture of the Pascal disk on the screen and then press and quickly release the mouse button. Like most operations on the Macintosh, this is far easier done than said.
- *Double-clicking.* This operation only differs from clicking in that you press and release the mouse button twice in quick succession after positioning the pointer.
- *Pressing.* To *press* something means to move the pointer to it and then hold down the mouse button without moving the mouse.
- *Dragging.* To *drag,* you position the pointer on an object; press the mouse button and, holding it down, move the mouse to another position; then release the button. The object to which you originally pointed will move to the new pointer location.
- *Menu selection.* A *menu* (a list of possible command choices) is displayed when you press one of the words or phrases in the *menu bar* at the top of the screen. To choose one of the commands, drag to the command you want and release the mouse button. For example, if you were instructed to "choose Go from the Run menu," you would (1) move the pointer to the word "Run" in the

menu bar, (2) press and hold the mouse button, (3) move the pointer downward in the displayed menu until the word "Go" was highlighted, and (4) release the mouse button. Again, this sounds more complex than it actually is.

- *Opening icons.* Application programs, your own programs and word processing documents, pictures, and other material stored on the disks are often represented as little pictures called *icons* on the Macintosh screen. Some icons are shown in Figure 1-1. Disks themselves are also represented as disk icons. The most common operation on these icons is to *open* them. Opening an icon creates a window through which you can view the contents of the icon. (What precisely happens depends on what the icon represents.) To open an icon, you *select* the icon by clicking it and then choose Open from the File menu. A faster method is simply to double-click the icon.

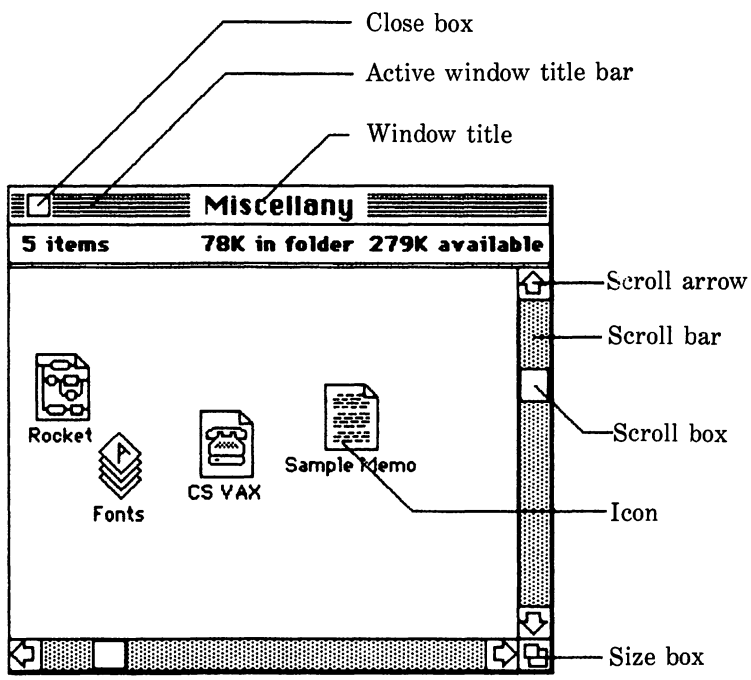


Figure 1-1.

Window anatomy

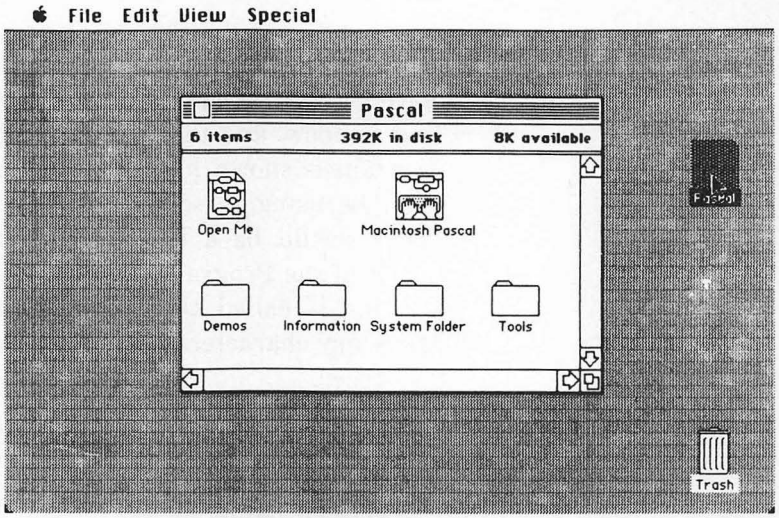
- *Making a window active.* There may be an arbitrary number of windows open on the Macintosh screen at one time. Most operations on windows only work on the *active* window. The active window is the one with the horizontal lines highlighting the title bar; Figure 1-1 shows a typical active window. To make a window active, simply click on a visible part of the window. An active window will overlap any other window on the screen.
- *Closing a window.* If you have created a window by opening an icon, you will often be able to close it again by making it active (if necessary) and then clicking the *close box,* which is in the upper-left corner of the window. (See Figure 1-1 for the position of the close box.)
- *Changing a window's size.* You may change an active window's size by dragging the window's size box (see Figure 1-1) to a new position. The window's upper-left corner will remain in the same position; only the position of the window's lower-right corner is changed.
- *Moving a window.* You can move the active window somewhere else on the screen by dragging its title bar (see Figure 1-1).
- *Scrolling the window.* If the active window is too small to display all the information in it, one or both of the scroll bars will change from an all white to a gray bar with a small white *scroll box* inside (see Figure 1-1). The white box indicates the relative position of the part of the window you can see with respect to the entire window. Bringing other parts of the window into view is called *scrolling* the window. You may do this in one of three ways: (1) Press one of the scroll arrows to move the small white box a short distance in the direction indicated. (2) Click in the scroll bar on either side of the scroll box; this will move the scroll box a windowful at a time. (3) Drag the scroll box to the desired position within the scroll bar. One of these methods will usually be more suited than the other two to any particular kind of movement within the window.

These essential operations are all you need to know in order to get started with Macintosh Pascal. Additional operations such as selecting, cutting, and pasting will be discussed as the need arises.

YOUR FIRST PROGRAM

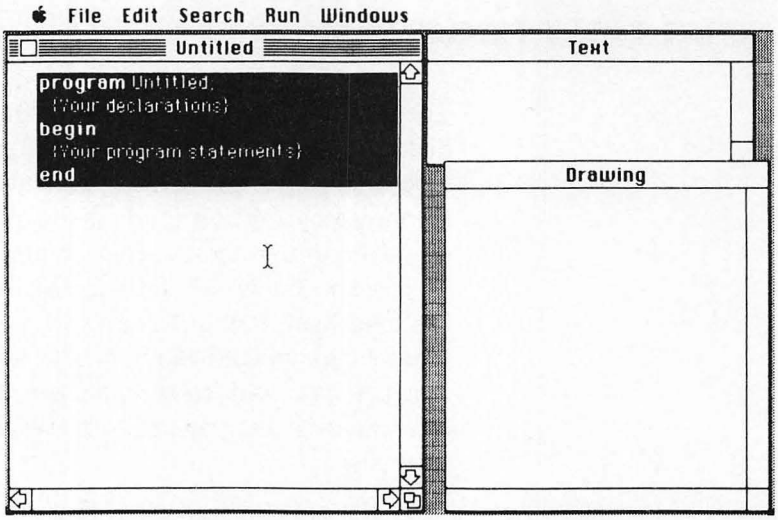
Turn your Macintosh on and then insert the Macintosh Pascal disk in the internal drive. After a good amount of disk whirring, you'll see the normal Macintosh desktop environment displayed on the screen. A disk icon labeled "Pascal" should be in the upper-right corner of the screen.

If your Macintosh Pascal disk has previously been used, the disk icon may automatically open to show the disk contents in a window labeled "Pascal." If only the disk icon appears, you will need to open it yourself: double-click the Pascal disk icon to open it. The results should look something like this:



The specific arrangement of icons in the window is not important, and your window may look slightly different. If someone has used the disk previously, there may be a different arrangement of the icons or even missing or extra icons. It is only important that the fingers-on-keyboard Macintosh Pascal icon be present somewhere in the window.

Double-click the Macintosh Pascal icon; this opens (runs) Macintosh Pascal. After some more disk whirring, you should see the following display.



When you start Macintosh Pascal, three windows are displayed: the Program window (here labeled "Untitled"), the Text window, and the Drawing window. A prototype Pascal program is shown highlighted in the Program window; we won't be using it, so press the BACKSPACE key to erase it. There should be a blinking vertical line in the upper-left corner of the Program window, and nothing else. This blinking line is called the *insertion point* because it indicates where any characters you type at the keyboard will appear on the screen.

You are now ready to work with your first Macintosh Pascal program. Here it is (but don't type it in just yet):

```
program hello;
{ My first Pascal program }

begin { hello }
 writeln('Hello there, world.')
end.
```

This program is intentionally short and simple. When run, it displays the following output:

```
Hello there, world.
```

Such a simple program will allow you to concentrate on the mechanics of entering, running, and modifying Macintosh

Pascal programs without worrying about the language itself. (There will be time for that later.)

Before you start entering the program, note the following rules that apply to typing Macintosh Pascal programs:

- Pressing the BACKSPACE key on the keyboard erases the character preceding the insertion point. If you notice a typing mistake, you can always press BACKSPACE until the error is erased; then simply continue typing from that point. This method of correcting typing errors always works, even if your mistake is a number of lines above your current insertion point, but it is most convenient for fixing mistakes closely preceding the insertion point. We'll discuss more advanced editing tricks in a bit.
- Don't worry about the boldfaced words **program**, **begin**, and **end** in the program listing. You don't need to type any special command to make these words boldface; Macintosh Pascal will automatically change them (and other special words) to boldface once it realizes they have been typed. These special words are called *reserved* words in Pascal.
- Don't worry about any special positioning of the individual program lines. For example, you'll notice that the line

  ```
  writeln('Hello there, world.')
  ```

  is indented from the lines below and above it. This indentation is supplied by Macintosh Pascal; you won't have to type it yourself.
- For now, press the RETURN key after every line you enter. (In certain cases, Macintosh Pascal will automatically skip down to another line, but you'll learn later how to use this.)
- Often you'll need to type spaces in a program to separate words from one another; in this program, you'll need to type at least one space between the word **program** and the word "hello."
- The marks on either side of the words

  ```
  Hello there, world.
  ```

  are apostrophes; the apostrophe key is found next to the RETURN key on the Macintosh keyboard. Don't confuse

the apostrophe (') with the double quotation mark (") or the grave accent (`).

Keeping these simple rules in mind, type in the sample program. Try to avoid spelling errors and other mistakes; remember, Pascal is very picky about innocent errors in punctuation and spelling. Once you finish, double-check your program to make sure it looks like the one here; if there are any differences, use BACKSPACE to change them.

Once you have your program typed in, choose the Go option from the Run menu. Many things will happen: the disk will whir, the pointer will momentarily change its shape into a *crosshair*, and most of the menu titles will dim briefly. But most importantly, the words "Hello there, world" should appear in the Text window. You have successfully written and run your first Macintosh Pascal program.

If something went wrong, you probably made some error in typing in the program. If you are a programming novice, it is easy to feel frustrated or confused at this point. Don't let these feelings discourage you; making mistakes is a fact of programming life.

If Macintosh Pascal detects a bug in your program, you'll see a message box describing the problem. After reading the message, click inside the box to make it disappear. Review the program entry rules previously listed; did you follow them all? Compare your program carefully with the one shown here. If you find the mistake in your program, use BACKSPACE to correct it, retype the remainder of the program, and choose Go from the Run menu once more. Eventually, you'll get things to work.

If you can't find the error by reading Macintosh Pascal's error message, reviewing the rules, and examining your program, you may want to skip ahead to the discussion of program errors and simple program editing in the next two sections, which should help. If all else fails, ask a friend for assistance; often others can detect problems that you have overlooked.

SIMPLE PROGRAM EDITING

Until now, we have only discussed one way to modify your program: pressing the BACKSPACE key until you erase what

you don't want and then retyping the remainder of the program from that point. A little thought should tell you that you will want far easier ways to make changes to your programs; you shouldn't have to erase and retype a large amount of program text in order to correct an error you made a number of lines earlier.

You will also want to modify your programs, both to correct errors and to make improvements to a working program. The process of typing in or changing a program is called *editing.* If you're like most programmers, most of the time you spend in front of the Macintosh will be devoted to editing your programs. In general, you will want to decrease the amount of time it takes to make program changes so you can spend more time writing programs.

Let's work on the Hello program to demonstrate how the simplest kind of program editing works.

The first thing you'll want to know is how to move the blinking vertical bar insertion point to another place in your program text. The answer will be familiar to anyone who has used a word processing program like MacWrite or has used the Note Pad desktop accessory: move the pointer to the spot in your program where you want to place the insertion point. (You move the pointer on the screen by moving the mouse on your desktop.) Note the pointer is the shape of an I-beam when it's inside the program window. When the pointer is where you want the insertion point to be, click the mouse button once; this will move the insertion point to the place indicated.

Use this technique to move the insertion point to between the "d" and the period in 'Hello there, world.' (If you don't get the insertion point to the desired spot, just try again.) Once the insertion point has been moved, press and release the BACKSPACE key; you'll see that it erases the character preceding the insertion point every time you press it, just as it did when you were entering the program the first time. (The BACKSPACE key automatically repeats if you hold it down, so don't let your attention wander.) Use BACKSPACE to erase the entire word "world," but don't erase anything else.

Let's make this program a little more personal. Type in your own name or that of a friend. Notice that each character you type is *inserted* into the text: all characters following the insertion point move over to make room for the new text you type. Your program will now look something like this:

```
program hello;
{ My first Pascal program }

begin { hello }
 writeln('Hello there, Colette.')
end.
```

Make sure you didn't accidentally erase the final apostrophe. (If you did, just BACKSPACE and retype until what you have matches what you see here.) Once more choose Go from the Run menu. Your program should display your new message in the Text window.

You have just learned three simple but vital program-editing techniques:

- *Insertion point movement.* To move the insertion point to a new position, just click the desired position.
- *Deleting text.* To delete one or more characters preceding the insertion point, press the BACKSPACE key one or more times.
- *Inserting text.* Typed characters are inserted at the current insertion point position.

You now know just about everything to make minor changes anywhere in your program: simply delete what you don't want and insert what you do want. For practice, change your program to read

```
program hello;
{ My first Pascal program }

begin { hello }
 writeln('Bonjour, guten Tag, and aloha, too.')
end.
```

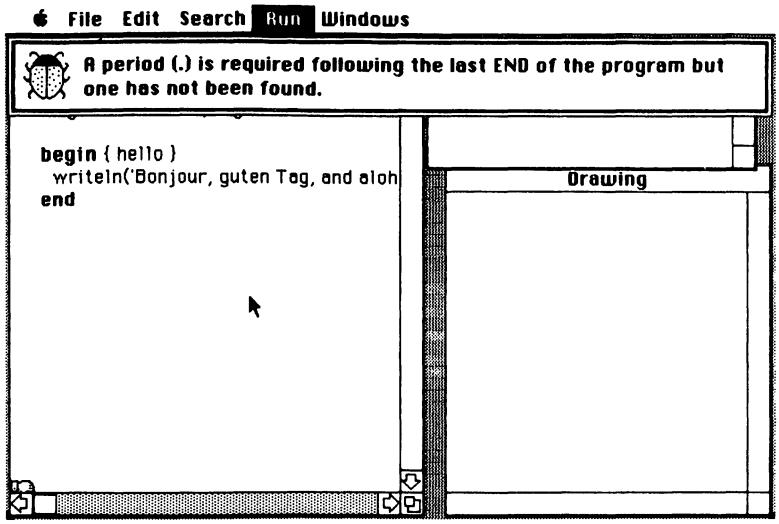
Notice the changed line is too long to fit into the Program window all at once; this will give you some practice using the scroll bar at the bottom of the Program window. When you're done, run your program by choosing Go from the Run menu.

SYNTAX ERRORS

If you have been lucky, you either have made no typing mistakes or have successfully corrected them before you tried to

run your program. In this section, we will create some errors in our program to see how Macintosh Pascal deals with such mistakes.

Move the insertion point just after the period following the word **end** on the last line of your program. Press BACKSPACE once to delete the period. Now try to run your program. You should see the following on your screen:



You now have a bug in your program; hence the insect picture in the box accompanying the error message. The error message is brief and to the point: you deleted the final period in the program, and that's why Pascal is refusing to run your program. Such an error is called a *syntax error* because it is a violation of the language rules of Pascal.

Notice also the picture of a hand in the left margin of your program at the bottom of the Program window. This is actually a "thumbs down" picture used by Pascal to tell you where it detected the syntax error. (Unfortunately you can't see the thumb, which is hidden by the bottom edge of the window.) Pascal not only tells you what kind of error you had; it also shows you where in the program the error was detected. In this case, Pascal searched for the period you deleted all the way down to the end of the window; this is why the hand is at the bottom of the window rather than up near the actual error.

You may get rid of the bug box by clicking anywhere within it. Once you have done that, put back the period follow-

ing **end** and run your program. Everything should now proceed normally.

At this point, experiment a little bit on your own to see what Pascal considers to be an error. You may not understand the reasons for the error, but you should try to see how Macintosh Pascal identifies errors for you. Try each of the following; after you see what happens in each instance, undo the change you made and go on to the next experiment. Note that with some of these changes you will see the thumbs down error indicator in its entirety. (Don't worry if the words in your program suddenly change to an "outline" font; that's part of Macintosh Pascal's error-detection scheme.)

- *Delete the* **o** *from the word* **program** *in the first line.* This produces a syntax error; Pascal always expects the word **program** at the beginning of a program.
- *Delete the space between the word* **program** *and* "hello." This produces the same error as before; Pascal requires at least one space between words to recognize them as separate words.
- *Change the word after* **program** *from* hello *to* greetings. This works fine; you have only changed the name of the program, and within certain restrictions, you may call your program by any name you want.
- *Delete the semicolon at the end of the first line.* This doesn't work. You'll remember that Pascal was picky about punctuation, and this is another example of how picky it can be.
- *Add an extra* **e** *to the word* **begin** *at any position you choose.* Again, this is a syntax error. Pascal expects the word **begin** at this point in the program; instead it finds something it considers to be a completely different word.
- *Delete the* "**r**" *from* "**writeln.**" Pascal fails to understand the word "witeln"; it is not part of the Pascal language, and you have not defined it.
- *Change* "writeln" *to* "write." Considering the previous experiment, you might not expect this to work. On the contrary, the program seems to work exactly as before. You'll see why later in the chapter.

After you have tried these changes, see if you can make legal and illegal changes of your own. Remember to restore your program to a working state if you make a change that Pascal considers to be unacceptable.

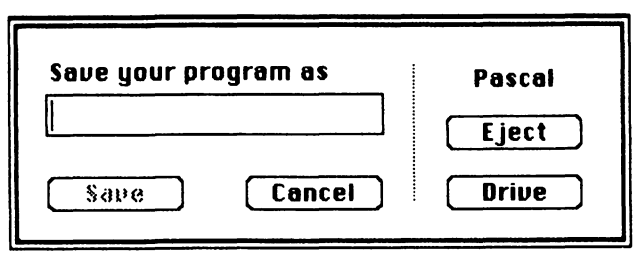
These experiments should convince you that Pascal can be a very demanding language; minor errors in punctuation, spacing, or spelling can stop your programs from working. Because Pascal (and other computer languages) demand perfection, not even experienced programmers entirely avoid syntax errors in their programs. As you continue programming, you can be certain that you'll get your share of bugs in the programs you write. (Be assured, however, that you will improve with experience.)

The good news here is that syntax errors are nothing special to worry about. Macintosh Pascal does its best to tell you what the error is and where it was detected. These clues will usually be enough to allow you to figure out the changes necessary for your program to work. By now you should be convinced that the mechanics of making changes to your programs will be easy.

PROTECTING YOUR WORK

The program you have written is temporarily stored in the computer's memory. We say "temporarily" because if you run another program or turn off the computer at this point, your program will be gone and unrecoverable. In order to save your program between sessions on the computer, you need to move it from temporary storage in memory to long-term storage on disk.

Saving your program on disk is rather easy. (First, however, make sure that you have undone all the changes from the previous section and that you have a working program.) Choose the Save as... option from the File menu. A *dialog box* will appear on your screen:



(If you do not have an external disk drive, the rectangle marked Drive will be absent. If an external drive is installed

but empty, the rectangle marked Drive will be dimmed.) The dialog box is often used in Macintosh Pascal (and other Macintosh software) to allow you to give commands, enter certain pieces of information, make choices, and set parameters. It's worthwhile to discuss the Save as... box in detail because it is typical of other dialog boxes you'll see while running Macintosh Pascal.

Your first step is to type a name into the rectangular area marked "Save your program as." There is a blinking insertion-point cursor in this space, indicating the point where any typed characters will be inserted. For now, type in the name "Hello"; use BACKSPACE to correct typing errors. Note that when you begin typing a name, the rounded rectangle marked "Save," which was dimmed, turns into a normal font.

In Macintosh terminology, the rounded rectangles in a dialog box containing words are *buttons*. To execute the command word contained in a button, simply click the button. Dimmed buttons represent commands that you cannot give at this point. (That's why the Save button was dimmed until you began typing a name under which to save your program.)

Another button in the Save as... box is Cancel; if you change your mind about saving your program, click this button.

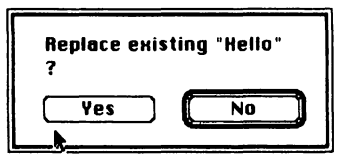
Do not store any programs on your Pascal disk; always use separate disks to store your programs. This will reduce wear and tear on your valuable Pascal disk.

Macintosh Pascal displays the name of the disk where your program will be saved on the right side of the box. There are also two buttons that allow you to save your program on another disk. The Eject button is useful if you want to save your program on a disk not currently in a drive. Clicking Eject causes the Macintosh to eject the current disk. If you have a single-drive system, you should eject the Pascal disk and then insert the disk on which you want to save your program. Macintosh Pascal gives appropriate prompts if the disk needs to be initialized.

If your Macintosh has an external disk drive and you haven't inserted a disk there, the Drive button will be present, but dimmed. Otherwise, you may click the Drive button to save your program onto the other disk. Clicking the Drive button acts as a toggle switch; you may click it again to switch back to your Pascal disk (although that is not recommended).

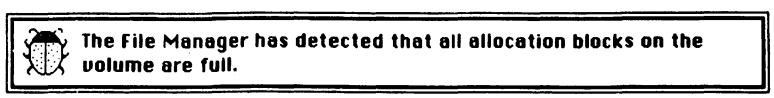
When you are convinced your program will be saved on the correct disk, click the Save button. After the disk spins for a while, the dialog box will disappear and you'll be back to Macintosh Pascal's "normal" state. (If you ejected the Pascal disk while saving your program, you'll be prompted to reinsert it.) Your program window should have changed its title from "Untitled" to "Hello."

There are two instances in which this would not have happened. Let's consider the simpler one first: If there had already been a program named Hello on the disk, you would have been warned about replacing a previously existing file:



If this happens, you have the choice of clicking either the Yes button or the No button. If you click the Yes button, Macintosh Pascal will save your program under the name Hello, destroying any program of that name previously existing on the disk. The recommended alternative is to click No (that's why the No button is emphasized); this sends you back to the Save as... box where you can choose another name or disk to save your program under. To do this, just click No, erase the name Hello using BACKSPACE, type in another name (your choice), and click the Save button. Unless you are perversely unlucky at choosing names, this should work.

The other case where Macintosh Pascal will not save your program is if you try to store it on a disk that has no room to accommodate it. In this case, Macintosh Pascal displays a bug box:



This is not one of Macintosh Pascal's clearest error messages; all it means is "Sorry, I can't do what you asked because the disk is full." If you get this error message, click

the bug box to make it go away. Then try the Save as... option again. This time, however, use the Eject or Drive button just described to save your program on a different disk.

All this detail on how to save your work may at first look a little daunting. In fact, everything discussed here takes very little time to do and becomes very natural. (Easier done than said, once more.)

To verify that your work has been saved, let's exit from Macintosh Pascal. Choose Quit from the File menu. After a moment, you'll be back where you were before you ran Macintosh Pascal.

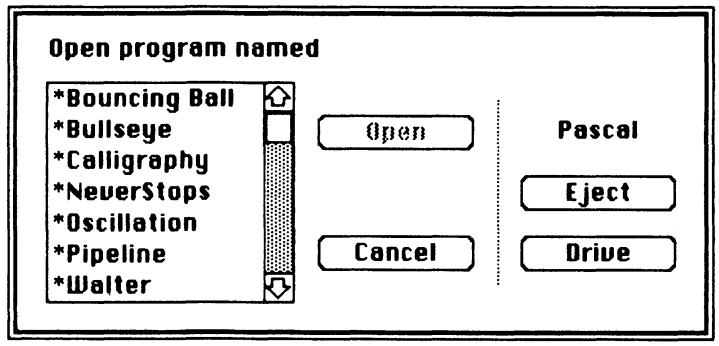
Close any open windows on the Macintosh screen by clicking their close boxes. Eject the Pascal disk by clicking its icon, and then choose Eject from the File menu. (If you have a second disk drive with a disk inserted, eject that disk as well, using the same method.) Turn off the Macintosh.

Had you neglected to save your program, it would have been erased from the computer's memory; if you wanted to run it again, you would have to type it in again. While this is no great hardship for simple programs like Hello, it could lead to problems when you begin to write larger programs. We'll see how to retrieve your program from storage in the next section.

RETRIEVING YOUR WORK

Turn your Macintosh back on and reinsert the Macintosh Pascal disk in the internal disk drive. Once more, you'll see the Pascal disk icon; double-click it, which will open the disk window. You'll remember that we previously double-clicked the Macintosh Pascal fingers-on-keyboard icon at this point to run Macintosh Pascal. Do that again.

After a few seconds, the normal Macintosh Pascal environment will appear with an Untitled program window. Before you retrieve your program you need to tell Macintosh Pascal that you will not be using this window. To do this, choose Close from the File menu; the Untitled window will vanish. Then choose the Open... option from the File menu. You should see something like the following dialog box:



This dialog box allows you to specify which program to bring into memory from disk. If necessary, you can retrieve programs from different disks by using the Eject and Drive buttons on the right side of the box.

- If you have a second disk drive, insert in it the disk on which you saved Hello. (If necessary, use the Drive and Eject buttons to remove any other disk currently in the drive.) Then click the Drive button to display the Pascal programs on that disk. If the disk contains other programs, Hello may not be visible in the window labeled "Open program named"; the window can only show seven program names at a time. If necessary, use the scroll bar in this window to bring Hello into view.
- If you have no second drive, click the Eject button to eject the Pascal disk from the internal drive. Insert the disk on which you saved Hello; you should then find Hello in the "Open program named" window.

Once you find the Hello program in the window, click the word "Hello." The word will be highlighted and the Open button will return to normal brightness. Now click the Open button. After a bit of disk reading, your program will be retrieved and displayed in its window, just as you typed it. (If you ejected the Pascal disk, you'll be prompted to reinsert as necessary.) If you want, run your program to convince yourself that it has really been returned to you safe and sound.

PRINTING YOUR PROGRAM

If you have a printer hooked up to your computer, you will often want to get listings of your Pascal programs. Let's see

how this is done. We'll assume in the following that you have successfully installed a printer on your system and that you have loaded the program you want to list into memory. If you have been following our discussion in this chapter, the Hello program is now in memory.

Choose the Page Setup option from the File menu. The following dialog box will be displayed:

You have a choice of four paper sizes on which to print your program:

- *US Letter*—standard 8 1/2 inches wide by 11 inches tall.
- *US Legal*—8 1/2 inches wide by 14 inches tall.
- *A4 Letter*—8 1/4 inches wide by 11 2/3 inches tall. (This is the standard size outside the U.S.)
- *International Fanfold*—8 1/4 inches wide by 12 inches tall.

The small hollow circles preceding each choice of paper size are called *check boxes.* One check box should contain a black dot; this indicates which paper size Macintosh Pascal currently expects you to be using. To specify another paper size (if necessary), click the check box next to the desired size.

You also have a choice of orientation:

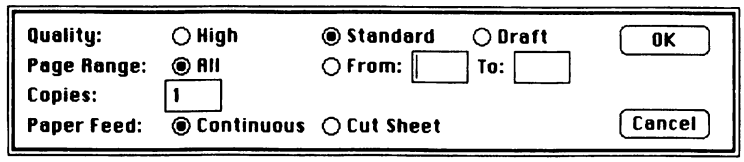
- *Tall*—Your program will be printed upright with the top line at the top of the page.
- *Tall Adjusted*—Same as Tall, but it correctly proportions graphics pictures from the Macintosh screen. Not applicable when you are printing text only.
- *Wide*—Your program is printed sideways on the page, the first line on each page going down the right side of the paper. (You should try this at least once.)

Click the check box that corresponds to the orientation you want. Normally, this will be Tall. Once you've chosen the two check boxes, click either the OK button to confirm the choices you made or the Cancel button to revert to the previous settings.

You only need to choose the Page Setup option if you want to change the previous setup; Macintosh Pascal remembers

the last decisions you made, even between sessions on the computer.

The next step is to choose Print from the File menu. You'll see a dialog box like this:



You have three choices for the "quality" of your program listing; click a check box for one of the following:

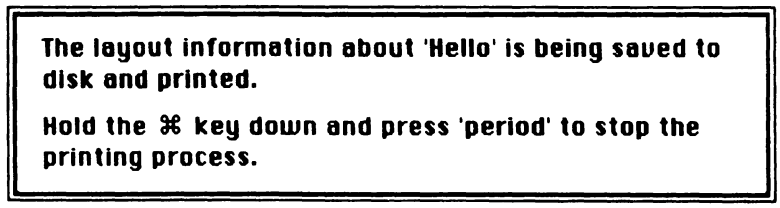
- *High*—Your program is printed in high-resolution mode. This is the slowest printing method, but the results are very attractive.
- *Standard*—Your program is printed with the same resolution as you see on the Macintosh screen. This is approximately twice as fast as the high-resolution mode.
- *Draft*—Your program is printed quickly, but the print quality is inferior to High or Standard quality. You may want to consider this choice when you are in a hurry and don't care too much about your listing's appearance.

You may also choose whether to print all of your program or only a certain page range. Click the check box for All to print the whole program. If you only want a range, click the check box to the left of the From: box. Then enter the numbers of the first and last pages you want to print in the From: and To: boxes by clicking in each box to set the insertion point and then typing in the desired numbers.

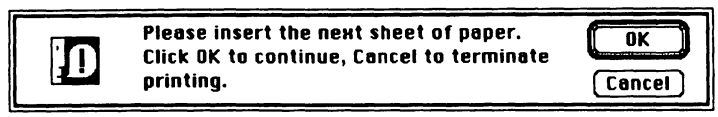
Normally, the number of copies specified in the Copies: box will be 1. To change it to something else, click in the box, use BACKSPACE to delete the old number, and type in the number of copies desired.

Finally, you may select a check box that governs the paper feed. Choose the Continuous check box if you are printing on fanfold or other continuous paper. If you are printing on single sheets, choose the Cut Sheet check box.

As soon as you have set up your selections, you may click either the OK button to begin printing or the Cancel button if you no longer want to print the program. If you click OK, you'll see the following self-explanatory box:

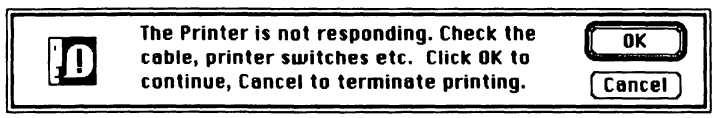


Your printer will then begin printing. If you selected Cut Sheet paper feed, you'll be prompted each time it's necessary for you to insert another sheet of paper.



After each page you have the choice of clicking OK to continue printing or Cancel to stop.

If your printer is turned off or is not ready to receive data, or if there is some other kind of hardware problem, you'll hear a warning beep and see the following dialog box:



If you see this box, check to make sure your printer is turned on. Make sure it is on-line; on the Apple Imagewriter, for example, the green SELECT light should be on. Check the cables to see if the printer is connected to the computer.

Another problem that can occur when you are trying to print is running out of space on disk. Macintosh Pascal saves information to disk before printing begins; if there's no room on the disk to store the information, the print is cancelled. (There's no dialog box to tell you what's going on, unfortunately; all you hear is a warning beep.) If this happens, you'll need to remove some files from the Macintosh Pascal disk; the methods for doing so are discussed in Chapter 3.

ANATOMY OF YOUR FIRST PROGRAM

So far we have not discussed the Pascal language at all; the program you typed in might have been just so many mumbo-jumbo magic words as far as you were concerned. The important thing to remember is that a program is simply a means of telling the computer what to do. The original program told the computer to perform the following action:

```
print the words "Hello there, world."
```

Unfortunately, the Macintosh can't understand English; more precisely, it can't take English commands and translate them into actions it can perform. That's why we need programming languages like Pascal. The Macintosh can understand the Pascal language; it will do what you want it to do if you can express your wishes in the form of a Pascal program.

Let's consider your first program line-by-line to see what makes it tick. Here is the first line:

```
program hello;
```

A line like this, known as the *program header*, will be the first line of every Pascal program you write. The word **program** tells Macintosh Pascal that this is the beginning of your program. The word **program** is a Pascal *reserved word;* you may not use it for any other purpose in your program. Figure 1-2 shows a list of all reserved words in Macintosh Pascal.

Macintosh Pascal Reserved Words

and	downto	if	otherwise	string	while
array	else	in	packed	then	with
begin	end	label	procedure	to	
case	file	mod	program	type	
const	for	nil	record	until	
div	function	not	repeat	uses	
do	goto	or	set	var	

Figure 1-2.

Macintosh Pascal reserved words

The word "hello" gives your program a name. The program's name must be a legal Pascal *identifier*, but otherwise it can be any name you want. Macintosh Pascal has the following broad rules governing what you can pick as an identifier:

- An identifier must begin with a letter.
- After the first letter, an identifier must contain only letters, digits, or the underline character. Any other characters are illegal in an identifier.
- An identifier must not be a Pascal reserved word.
- An identifier must have between 1 and 255 characters.

Identifiers are used for many other things in Pascal programs besides program names, so these rules will be important for you to remember.

All characters in a Macintosh Pascal identifier are *significant;* even if you have two 255-character identifiers in your program that only differ in their 255th character, Macintosh Pascal will not get them confused. (You, on the other hand, probably will.)

Macintosh Pascal also ignores the case of the letters in an identifier. You may spell the identifier "apple" as "Apple", "APPLE", or even "aPpLe". Pascal considers all of these to be the identifier "apple."

To continue the examination of your program, note that the semicolon at the end of the first line after the program name is mandatory. Pascal requires certain punctuation in certain spots, and this is one of them. A quick aside: many versions of Pascal would require you to write the first line of your program something like this:

```
program hello(input, output);
```

In Macintosh Pascal, the

```
(input, output)
```

words are optional. They don't bother anything if you put them in, but it's recommended that you omit them. It's just one less thing you need to remember—a blessing when you're trying to learn a difficult subject.

Now consider the next line

```
{ My first Pascal program }
```

This is called a *comment.* Pascal ignores any characters typed between a left brace ({) and a right brace (}). Therefore, comments are entirely irrelevant to the operation of the program. A program with comments does exactly the same thing as the same program with comments deleted. Although comments don't affect the computer, they can be useful to you. The comments are explanatory notes that can be read days, weeks, or months later when you or others are trying to figure out the inner workings of a dimly remembered program.

Macintosh Pascal allows alternative symbols for comment begin and end markers: you can use (* instead of { and *) instead of }. Macintosh Pascal insists that all comments you type in your programs be either on a line by themselves or at the end of a line. If you type a comment in the middle or beginning of a program line, Macintosh Pascal will move it to the end. And if, while typing a comment, you press the RETURN key before you type the right brace, Macintosh Pascal will supply the closing brace for you as well as an opening brace on the next line. Even if you are an old hand at Pascal, you'll no doubt want to experiment with this yourself. The formal rule for comment placement is that you can put a comment anywhere you can legally put a space, and vice versa.

The third line in our program (after the comment) is blank. Although Pascal is extremely picky about punctuation and spelling, it will allow you to have as many blank lines in your program as you want, wherever you want. Use blank lines to separate major portions of your program; this makes your program easier to read and understand. (You'll see more complex examples later.)

The next line signals Pascal that we are about to specify the "action" part of the program:

```
begin { hello }
```

The reserved word **begin** says that what follows is the executable part of the program: Pascal *statements* that will tell the computer "what to do." The comment { hello } is ignored by Pascal; it serves to remind whoever is reading the program that "**begin** marks the beginning of the executable

part of the Hello program." This fact may seem too obvious to deserve a comment in such a small program. But in larger programs where the **begin** and the **program** will be separated by many lines, it's worthwhile to clearly label the start of the executable part of the program.

We just said that the executable part of a Pascal program follows the word **begin**. In this case, the executable part of the latest version of Hello consists of this single statement:

```
writeln('Hello there, world.')
```

Pascal uses the *writeln* command to print out information. If you've been paying attention to what you have been doing, you may have speculated that writeln would print anything you put between the apostrophes. This is just about correct. In Pascal, any number of characters typed between apostrophes is known as a *character string* (or, informally, just as a *string*). To print out any string in Pascal, you simply put it inside parentheses following the writeln command.

Be careful, however, in printing a string containing an apostrophe like this one:

```
writeln('This isn't Jack's car.')
```

Try it. Delete the current string in the program and replace it with this one. Then try to run the program. The result is another bug box signaling that something is amiss. The problem is that Pascal thinks that the first embedded apostrophe ends the string; when it tries to make sense out of what follows, chaos results.

The correct solution is to type in two apostrophes for each actual apostrophe in the string:

```
writeln('This isn''t Jack''s car.')
```

Fix and test your program by inserting the additional apostrophes and trying to run it once more.

Another thing one might reasonably expect to work is the following "natural" method of writing on two lines at once:

```
writeln('This isn''t
Jack''s car.')
```

One might expect this to give the following output.

```
This isn't
Jack's car.
```

Test this by inserting a return after the word "isn"t" in the string and then trying to run the program. Again, Macintosh Pascal cannot make sense out of your program. It is against the rules of Pascal to continue a string onto another line by inserting a return.

If you want your output on two lines, you need to issue two writeln commands. Try your hand at modifying your program so that these lines appear between the **begin** and the **end**:

```
writeln('This isn"t');
writeln('Jack"s car.')
```

Note the required semicolon at the end of the first writeln. Once you have done this, run the program; this one should work.

This is a program containing two statements. The reason a semicolon was necessary between the two writeln statements is that, under the syntax rules of Pascal, a semicolon is necessary between any two statements. In the jargon, semicolons act as "statement separators" in Pascal. The **begin** and the **end** don't count as statements, however; a semicolon is never needed either after **begin** or before **end**.

If you carried out the experiments suggested in the section called "Syntax Errors," you may remember that changing the writeln statement to "write" seemed to have no effect whatsoever on your program; the program still worked and the output was indistinguishable from that produced using writeln.

Try the same experiment on the program you currently have in memory; change the first writeln command to a write:

```
write('This isn"t');
writeln('Jack"s car.')
```

The result is the message "This isn'tJack's car." all on one line with no space between the words "isn't" and "Jack's." A little thought should tell you the difference between write and writeln: writeln skips down to the beginning of the next line after writing, and write doesn't. (How would you insert a space to correct the words run together in the output?)

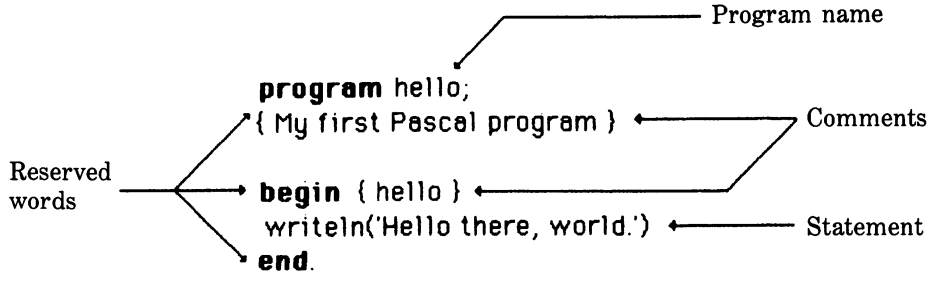


Figure 1-3.

Pascal program anatomy

The last line in your program is **end.** This tells Pascal that there are no more statements in your program; you should think of it as being paired off with the **begin** above it. To remind you of what you learned earlier in the chapter, the period following the **end** is required by Pascal.

Figure 1-3 is a summary of Pascal program structure you've learned so far.

For a final review of what you've learned about editing and about the Pascal language, modify your program to look like this:

```
program row;
{ write vs. writeln }

begin { row }
  write('Row, ');
  write('row, ');
  writeln('row your boat,');
  write('Gently down the ');
  writeln('stream.')
end.
```

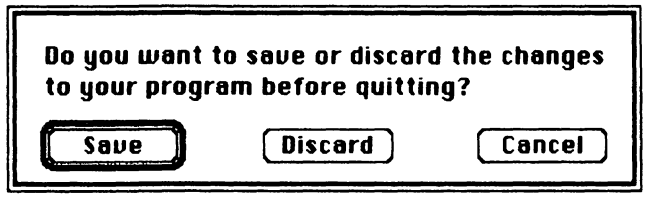
Try to predict what this program will do before you run it.

WINDING UP

Macintosh Pascal is protective of programs you have typed in. If you do not have a current version of the program in

memory saved on disk, Macintosh Pascal will not allow you to exit until you have verified that you really want to discard the version of the program in memory. You will be given another chance to save the program.

To see how this works, choose Quit from the File menu. If you have been following along in the chapter and have a changed version of the Hello program in memory, you'll see the following dialog box:



This dialog box gives you the following choices:

- *Save the program in memory.* If you click this button, the new version of your program will be saved, replacing the old version; you'll then be returned to the Macintosh desktop environment. (If there was no previous version of your program, you will be taken through the Save as...dialog described previously.)
- *Discard the program.* Clicking this button simply returns you to the Macintosh desktop; the original version of your program, if any, remains intact on the disk.
- *Cancel the Quit command.* Click this button if you change your mind about leaving Macintosh Pascal.

Once you leave Macintosh Pascal, you may either run other applications or turn off the computer. Don't forget to eject all disks from their drives before you turn off the computer.

VARIABLES AND LOOPS

2

Programs, after all, are concrete formulations of abstract *algorithms* based on particular representations and structures of *data*.

N. Wirth
Algorithms + Data Structures = Programs
(Prentice-Hall, 1976)

In this chapter we'll investigate more capabilities of the Pascal language. So far we have only seen how to write simple programs that output whatever character strings we tack on to write and writeln commands. This capability, while important, only plays a minor part in most programs. Your next step in learning Pascal is to learn to store information in the computer's memory using *variables* and to tell the computer to execute certain commands repetitively using *loops*.

UNDERSTANDING VARIABLES

Every significant program you write will require the computer to "remember" certain pieces of information that the program will need to use while it is running. These pieces of information are called *variables* in nearly all computer languages, including Pascal. They are called variables because they may vary during the time your program is running; the information contained in a given variable may take on many

different values based on the actions specified in your program.

It may be useful for you to think of a variable as a box hidden away somewhere inside your computer. This image is fairly accurate for describing the function of a variable; it is a storage location, which should always be thought of as containing something. (If you know a little about computers, you may recognize that this also describes the computer's memory.)

If you want to use a variable in your program, Pascal makes the following demands:

- You must specify a name by which your program can refer to the variable.
- You must tell Pascal what type of information you will be storing in the variable.

Giving each variable a name distinguishes it from all the other variables in your program; this means that if your program needs two different variables, you must give them two different names. The names you give to your variables must follow the rules for identifiers given in Chapter 1.

You must specify the type of variable so that Pascal can set aside an appropriate amount of the computer's memory to store the information contained in the variable. To use the box analogy, Pascal has many different kinds of boxes available, each of a different size, shape, and color. The type of variable will determine which kind of box Pascal will use to store the information represented by the variable.

Specifying a name and a type for a variable is called defining or *declaring* the variable. Pascal sets aside a special place for you to define your variables, called the *variable definition part* of your program. A variable definition part must come after the program header line and before the word **begin** that leads off the executable statements in your program. (You'll be given more precise parameters later.)

A typical variable definition part might appear in a program as follows:

```
var
 page_length : integer;
 width : real;
 done : Boolean;
 digit : char;
 line : string;
 hue : color;
```

Informally, this tells Pascal the following: This program will use the variables page_length, width, done, digit, line, and hue. The variable page_length will be of type integer, the variable width will be of type real, the variable done will be of type Boolean, the variable digit will be of type char, the variable line will be of type **string**, and finally, the variable hue will be of type color. Experienced programmers will make this even less formal, saying, for example, "done is a Boolean." This is fine, as long as you remember that such a shorthand way of discussing a program often leaves out some important words.

If you want to define more than one variable of the same type, you may use a list of variable names separated by commas in each line of the variable definition part instead of just a single variable name. For example, the following variable definition part declares the real variables x, y, and z, and the integer variables i, j, and k:

```
var
 i, j, k : integer;
 x, y, z : real;
```

Pascal doesn't care whether you define your variables in list form or in the form of one variable per line; nor does it care what order you define them in. The important thing is that all the variables in your program *get* defined somehow.

Every variable definition part you write in your programs will follow the general form shown in Figure 2-1. Pascal requires that reserved word **var** lead off the part followed by one or more variable lists, each with its own type. There must be a colon between each variable list and the corresponding type, and the type must be followed with a semicolon.

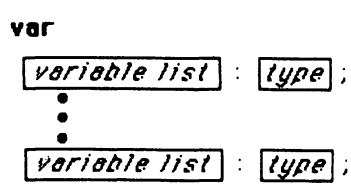


Figure 2-1.

Variable definition part syntax

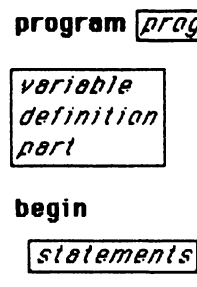


Figure 2-2.

Pascal program syntax (incomplete)

Figure 2-1 is an example of a *syntax sketch*, an informal semipictorial description of the syntax rules of a particular facet of Pascal. We will make use of such sketches as the need arises. For example, Figure 2-2 is a syntax sketch showing everything we have learned so far about the overall structure of a Pascal program.

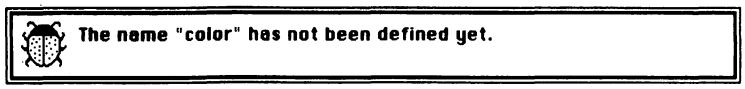
Let's now experiment with variable definitions. Turn on your Macintosh and open Macintosh Pascal. As before, the prototype Pascal program will be highlighted in the Program window; press the BACKSPACE key to erase it. Then enter the following program:

```
program variable_lab;
{ Experiments with variables }

 var
  page_length : integer;
  width : real;
  done : Boolean;
  digit : char;
  line : string;
  hue : color;

begin { variable_lab }
end.
```

It may seem strange to enter **begin** and **end** with nothing between them. In fact, it is entirely legal to do so in Pascal; it

simply means the program won't do anything when it is run—and that's fine for now.

Try to run this program; assuming you typed it in as shown, you should get this bug box:



This error message is especially revealing; it tells us a number of important things about Pascal.

First, the error message is very specific. You tried to define a variable of type color without first defining what you *meant* by the word color. This implies a simple, elegant rule you will always have to follow in writing Pascal programs:

In Pascal,
any identifier must be defined
before it can be used..

If you are a beginning Pascal programmer, you should write this rule down (in large letters) and put it in a place where you'll see it often.

Second, this error message tells you that you *didn't* define the type color. It did not say you *couldn't* define the type color; in fact, you can define your own types in any Pascal program and give those types whatever names you want. The method will be introduced in Chapter 10 where you will define the type color, as well as other types.

Finally, notice where the error occurred in the program: only *after* the declarations of the first five variables. This means that as far as Pascal was concerned, those first five variables were of types already defined. To verify this, simply delete the offending line containing the undefined type. (Remember, you must click the bug box before you can continue working.) Then try to run the program. You should now have no syntax errors; it is a valid Pascal program. It does what you programmed it to do—nothing.

You have already seen the word **string** as one of the reserved words of Macintosh Pascal. The remaining four types, integer, real, Boolean, and char, are all predefined types built into Pascal. The integer, real, and Boolean types

are discussed in this chapter, the char and **string** types in Chapter 6.

Macintosh Pascal has a number of other built-in or predefined identifiers in addition to the ones just mentioned; they will be discussed as they arise.

THE INTEGER TYPE

In mathematics, an integer is any number without a decimal point. In Pascal, the integer type is used to represent these "mathematical" integers. Let's investigate how the integer type can be used in Macintosh Pascal. Use the deletion and insertion techniques discussed in Chapter 1 to make the variable_lab program look like this:

```
program variable_lab;
{ Experiments with variables }

 var
  i, j, k : integer;

begin { variable_lab }
 i := 324;
 j := 23;
 k := i + j;
 writeln('The value of i is:', i);
 writeln('The value of j is:', j);
 writeln('The value of k is:', k)
end.
```

As usual, try to type your program so that it is identical to the one shown here. In particular, you'll want to be sure you type the two-character := symbol as a colon followed by an equal sign with no spaces between the two. Run the program; this output should appear in the Text window:

```
The value of i is:      324
The value of j is:       23
The value of k is:      347
```

Let's now consider the executable part of the program that follows **begin**. The first two statements are examples of the very important *assignment statement.* An assignment state-

ment is used in Pascal for a single purpose: to give a variable a value. The net effect of the two statements

```
i := 324;
j := 23
```

is to give the value 324 to the variable i and the value 23 to the variable j. To use the box image once again, 324 is stored in the box labeled i, and 23 is tucked away in the box that has the label j.

(Remember that the function of semicolons in the executable part of the program is to separate one statement from another. When you are shown program segments in this book, the semicolon used to separate the program segment from what follows will be omitted.)

The next statement is also an assignment statement, but it differs slightly from the other two:

```
k := i + j
```

If we were to explain the action this statement performs, we might say, "Variable k gets the value obtained by adding together the contents of variable i and variable j." (The plus sign indicates addition in Pascal.)

It is important that you understand *exactly* what happens when an assignment statement is executed, since assignment statements are extremely common in Pascal programs. The statement specifies that an action is to occur: the variable named on the left side of the := symbol takes on a new value, replacing whatever previous value it had. The new value is specified on the right side of the := symbol; we've seen the right side can either be a single value (324 or 23, for example) or a mathematical operation between two variables (i + j, for example).

Pascal calls the right side of an assignment statement an *expression.* The syntax sketch of the assignment statement is shown in Figure 2-3. The actual definition of an expression

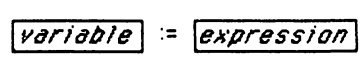


Figure 2-3.

Assignment statement syntax

comes later in this chapter; for now let's concentrate on some more examples.

Many beginning programmers are confused by the equal sign in the assignment statement. In algebra, the equal sign represents not an action, but a fact. To the student of mathematics, the equation

```
x = x + 1
```

is obvious nonsense: it can't be satisfied for any possible value of x.

In Pascal, however, the assignment statement

```
x := x + 1
```

makes perfect sense; in fact, you'll see similar statements all the time. It means "take the value contained in the variable x, add 1 to that value, and store the result back into the variable x." The net effect of the statement, then, is to add 1 to the value of x. This is called *incrementing* x.

It is also possible to write legal algebraic equations that would make no sense in Pascal. For example, this might be a sensible thing to see in a math class:

```
x + y = 3
```

But the equivalent assignment statement,

```
{ THIS IS ILLEGAL IN PASCAL }
x + y := 3
```

makes no sense whatsoever to Pascal; in all legal assignment statements there must be a single variable on the left side of the := symbol to receive the value from the right side.

For these reasons, you should pronounce the := symbol as "gets" when reading programs aloud: "x gets x plus one," for example. This emphasizes the non-mathematical nature of the assignment statement.

Look again at the final three statements of our variable_lab program:

```
writeln('The value of i is:', i);
writeln('The value of j is:', j);
writeln('The value of k is:', k)
```

These are writeln statements, slightly different from those you've seen before. Here, each writeln displays two things: the string "The value of i is:" and the value of a variable. Compare the output of the program shown in the Text window with the value of the variables i, j, and k; you should verify that writeln actually does write out the variable value in each case.

In general, a writeln statement (and write as well) can be used to output the value of any number of items in a single statement. Simply put each one into a comma-separated list, and put the list between parentheses following write or writeln. (Write and writeln will be explored more thoroughly later in this chapter.)

For now, let's carry out some experiments on the variable_lab program. Change the line

```
k := i + j
```

to the following:

```
k := i - j
```

by deleting the plus sign and inserting a minus sign in the same spot. Then run the program. The value of k changes to 301, showing that the minus sign indicates subtraction, just as the plus sign indicated addition.

Now try replacing the minus sign with an asterisk; the statement should now read:

```
k := i * j
```

When you run the program, the value of k should be displayed as 7452, or the result of multiplying 324(i) times 23(j). The asterisk indicates a multiplication operation.

You can't put just any character between the i and j; for example, try replacing the asterisk with a percent sign:

```
k := i % j
```

When you run the program you'll get a bug box:

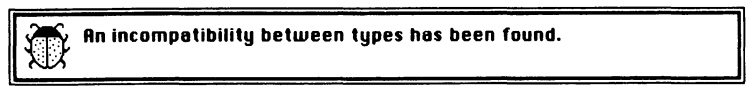
This does not make sense as a statement.

This nonspecific error message means Pascal can't make any sense out of what you have typed in as a program statement. The percent sign doesn't mean anything to Pascal.

Now try replacing the percent sign with a slash:

```
k := i / j
```

When the new version of the program is run, the result is a bug box you may not have expected:



The problem is not that Pascal can't make sense out of the slash character. After all, it had to have made some sense out of it, or it would have given the same message it gave for the percent sign. The problem is that the slash causes i to be divided by j. In general, dividing one integer by another will not give another integer; instead, you will get a number with a fractional part. Pascal is complaining that you are trying to fit a number that might have a fractional part (the result of i/j) into a variable whose assumed type does not allow fractional parts (the variable k). Trying to do this in Pascal makes as much sense as storing sugar in a ketchup bottle.

Pascal provides an integer division operator called **div**. To see how it works, replace the slash in the program with **div**:

```
k := i div j
```

When you run the program, it should display 14 for the value of k. (Since **div** is a reserved word, Macintosh Pascal automatically boldfaces it in your listing.) The **div** operator carries out the division of two integers and discards any fractional part, representing the result as an integer.

Another operator associated with **div** is **mod**. Try the program once more, replacing **div** with **mod**:

```
k := i mod j
```

The result obtained for k should be 2. The **mod** operation gives the integer remainder obtained when dividing i by j (that is, 23 into 324 goes 14 times with 2 remaining).

Don't worry too much if you don't see any immediate prac-

tical use for these last two operations; in fact, the addition, subtraction, and multiplication operations on integers are usually more common. But **div** and **mod** also have their uses, as you'll see later.

In mathematics there are an infinite number of integers; there is no "biggest" integer. Due to memory and speed considerations, however, Pascal puts limits on the possible range of an integer variable. The largest integer value in Macintosh Pascal is 32767.

There are also negative integers, both in mathematics and Pascal. In Pascal, an integer value can go all the way down to −32767; there are as many possible negative integer values as positive. (Zero is considered to be neither positive nor negative.)

To write integer values in Pascal, you must follow a few simple rules:

- Don't put commas in the value (write 10234, not 10,234).
- Negative values are preceded by a minus sign (for example, −345).
- For Macintosh Pascal *only*, you may write integer constants in hexadecimal (base 16) instead of decimal (base 10). To signal Macintosh Pascal that an integer constant is being written in hexadecimal, precede the value with a dollar sign. For example, the assignment

```
i := $1fa
```

  is equivalent to the assignment

```
i := 506
```

You should become familiar with the basics of integer arithmetic. Be sure you can predict the results of any simple addition, subtraction, or multiplication operation. See if you can change the values assigned to i and j in the variable_lab program and predict the result for k before you run the program.

In addition, you might want to try the following experiments:

- What happens if you try to **div** by 0? How about **mod** when the second integer specified is 0?
- How do **div** and **mod** behave when one or both integers operated on are negative?

- What happens if your program specifies an operation that would give a result outside the legal integer range, −32767 to 32767?

THE for LOOP

So far you have seen that the statements between **begin** and **end** are all carried out in the order in which they appear. Each statement is executed once. This is known as *sequential execution,* and it is a fundamental concept in most programming languages.

Sequential execution is not flexible enough to write any but the simplest programs, however. Most programs require the ability to carry out *conditional execution:* to perform different actions based on the current value of a variable or some other condition. Also necessary is the ability of the program to do *repetitive execution:* to perform certain actions over and over, either a fixed number of times or as long as a certain condition holds true.

These three types of program execution (sequential, conditional, and repetitive) are known as *control structures.* Just about any kind of program can be put together by specifying the desired combinations of these three types of control structures. Most of what is discussed about Pascal in the remainder of this chapter and in Chapter 4 will be about how to put these structures together.

The simplest repetitive control structure is one that performs an action (or group of actions) a specified number of times. In Pascal, this is written using the **for** loop. To see how it works, first clear the computer's memory of any previous program, if there is one. (One easy way to erase an entire program is to choose Close from the File menu, saving your program if you want, and then choose New from the File menu.) Once you have cleared away any previous program, enter the following:

```
program for_lab;
{ Experiments with for loops }

var
 i : integer;
```

```
begin { for_lab }
 writeln('I hear you knocking.');
 for i := 1 to 4 do
  write('Knock ');
 writeln;
 writeln('But you can''t come in.')
end.
```

When you run the program, you should get the following output:

```
I hear you knocking.
Knock Knock Knock Knock
But you can't come in.
```

It should be clear that **for** specifies that the write statement is to be done four times. Change the 4 into a 10 and rerun the program. (You'll have to expand the Text window to see all the output in this case.) This should convince you that you can stipulate just about any number of repeats of the loop, at least within the limited range of the integer type.

The loop limits do not have to be constant numbers. Make the following minor modifications to the program:

```
program for_lab;
{ Experiments with for loops }

 var
  i : integer;
  initial, final : integer;

begin { for_lab }
 initial := 1;
 final := 4;
 writeln('I hear you knocking.');
 for i := initial to final do
  write('Knock ');
 writeln;
 writeln('But you can''t come in.')
end.
```

This will give the same output as before.

The variable used after the word **for** (which is i in this case) is called the *loop control variable.* The loop control vari-

able takes on successive values each time through the loop. To see this, change the program slightly:

```
program for_lab;
{ Experiments with for loops }

var
 i : integer;
 initial, final : integer;

begin { for_lab }
 initial := 1;
 final := 4;
 writeln('I hear you counting.');
 for i := initial to final do
  write(i);
 writeln;
 writeln('But you can''t count on me.')
end.
```

Your output now looks like this:

```
I hear you counting.
        1       2       3
4
But you can't count on me.
```

Thus two different but related things happen in the **for** loop. First, the loop is executed a known number of times. Second, the loop control variable takes successive values on each repetition of the loop. When you write your own programs, one or the other and often both of these features will be important to you.

The initial value of the loop control variable doesn't have to be 1. Try changing the lines

```
initial := 1;
final := 4;
```

to these:

```
initial := 13;
final := 25;
```

The output from this version of the program is proof that you can specify any integer values for the initial and final values of the loop control variable. Go ahead and make a few changes of your own to the numbers assigned to the initial and final values; include negative values as well as positive and zero values.

If you haven't done so already, try an initial value greater than the final value; for example,

```
initial := 13;
final := 10;
```

In this case, the loop is executed zero times—in other words, not at all. An initial value for the loop control variable that is greater than the final value may not make a lot of sense to you right now. However, you won't always know at the time you write a **for** loop what the limits will be; you won't always be sure the initial value is less than the final value. In such cases, you'll want to know that the loop may not be executed at all.

Pascal also provides a version of the **for** loop that counts backward; just replace the word **to** with **downto**. Try modifying the program as follows:

```
program for_lab;
{ Experiments with for loops }

 var
  i : integer;
  initial, final : integer;

begin { for_lab }
 initial := 10;
 final := 1;
 writeln('Countdown:');
 for i := initial downto final do
  write(i);
 writeln;
 writeln('Liftoff!')
end.
```

To see the output this produces, you may have to enlarge or scroll the Text window. (The actual form of the output

depends on the horizontal size of the window.) The result of using **downto** should appear like this:

```
Countdown:
      10        9        8
7        6        5        4
3        2        1
Liftoff!
```

When you use write, Pascal will print as many numbers as it can fit on a single line of the Text window and then skip down to the next line and continue output. Why are the numbers printed here so far apart? Unless you specify differently, the write and writeln statements use a *field width* of 8 when printing integers. This means they will use at least eight spaces to print an integer value; if the number itself requires fewer than eight characters, it is *right-justified* (padded with blanks on the left) so that there are eight characters printed in all.

If you don't like this default field width, it is easy to change it. In a write or writeln statement, follow the expression being printed with a colon and a number specifying whatever field width you want. For example, in the for_lab program, try inserting a field width parameter in the write statement as follows:

```
write(i : 2)
```

Now when you run the program, you'll get the following result:

```
Countdown:
10 9 8 7 6 5 4 3 2 1
Liftoff!
```

Since you specified a field width of 2, all your numbers are printed using at least two spaces. Try changing the field width to 1. This should be the result:

```
Countdown:
10987654321
Liftoff!
```

If a number requires more room to be printed than the field width specifies, it is printed out with no padding.

Here are some questions for you to answer by experimentation:

- Does it make any sense to have a field width of 0 (or less)? What happens when you try it?
- Does the field width you specify have to be a constant? To find out, try replacing the write statement inside the loop with

  ```
  writeln(i : i)
  ```

  Run the program. If you haven't already, expand the Text window vertically to see all the output. Can you explain what's going on? (Remember, you changed the write to a writeln.) Once you have figured it out, try to predict what would happen if the write statement looked like this:

  ```
  writeln(i : 11 - i)
  ```

  After you predict the output, see if you were right by changing and running the program. If you were wrong, try to find out why.

The syntax sketch for a **for** statement is shown in Figure 2-4. Once more we are using the term *expression* as part of the definition; remember, we haven't explicitly said what that word means as yet, but we've given a number of examples. Also note that the horizontal bar indicates that you may use either the **to** or **downto** in a **for** statement, but not both.

Figure 2-4 also tells you that only a single statement can be inside the loop. This may make you doubt the usefulness of the **for** statement. What if we want to perform more than one action inside the loop?

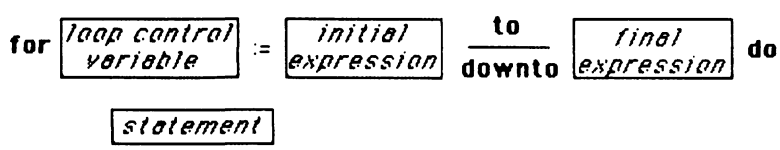


Figure 2-4.

for statement syntax

Figure 2-5.

Compound statement syntax

Pascal has a simple construction that allows you to get around the one-statement rule by grouping statements together. This construction is called the *compound statement*, and its syntax is shown in Figure 2-5. A general rule in Pascal is that you may write a compound statement anywhere you can write a single statement. (As far as the language rules of Pascal are concerned, a compound statement is just another kind of statement.)

Let's see how a compound statement is used in a program. Try modifying the for_lab program to look like this:

```
program for_lab;
{ Experiments with for loops }

 var
  i : integer;
  initial, final : integer;

begin { for_lab }
 initial := 10;
 final := 1;
 writeln('Countdown:');
 for i := initial downto final do
  begin
   writeln(i : 11 - i);
   sysbeep(45)
  end;
 writeln;
 writeln('Liftoff!')
end.
```

Now there are two things inside the **for** loop: a writeln statement as usual and also a command called sysbeep. Rather than trying to guess what will happen, run the program to find out.

Two things happen with each execution of the loop. The number in the countdown is printed as before, but now it's followed by a beep. (If you don't hear any sound, the speaker volume has probably been set to zero; for the purposes of testing this program, use the Control Panel desk accessory to adjust the speaker volume to an audible level.) When the sysbeep statement is executed, it makes the Macintosh's speaker ring for about a second; this also has the effect of slowing down the loop so that it seems more like a real countdown.

The Macintosh's sound capabilities are discussed in more detail in Chapter 8, but for now you might want to try putting different numbers in the parentheses following the word sysbeep. What does changing the number do to the sound? What happens if you replace the number with the expression 10*i?

There are a few more facts about the **for** statement that are easy to discover via experimentation. For example, what happens if you try to change the final limit inside the loop? Will this change the number of times the loop is executed? Without altering the rest of the program, try changing the **for** statement to look like this:

```
for i := initial downto final do
 begin
  writeln(i : 11 - i);
  final := 5;
  sysbeep(45)
 end
```

You might speculate that changing the value of final to 5 inside the loop would make the **for** loop only count down to 5. But nothing of the sort occurs: the loop proceeds as before. The rule is that both the initial and final values of the loop control variable are evaluated once, and once only, before the loop starts. Thus, the loop will always be executed up or down to that previously determined final value.

There's no way to alter the value of the loop control variable inside the loop. To prove it to yourself, you can try it by adding this statement anywhere inside the loop:

```
i := 1
```

Running the program gives you this precisely worded bug box:

The moral to be extracted from these last two tests is that the **for** statement is only useful for performing actions a definite number of times; attempts to force it into some other behavior by altering the loop limits or jiggling the loop control variable are doomed to failure.

When discussing the **for** statement, most Pascal texts solemnly intone something to the effect that "after completion, the value of the loop control variable is undefined." This means that Macintosh Pascal sets the loop control value to an unpredictable value on exiting from a **for** statement. To illustrate, first remove the bug we introduced into the for_lab program and then add a line to our program after the loop that prints out the value of i, the loop control variable:

```
writeln('Outside loop, i is ', i : 1)
```

Run the program two or three times to make sure of the results. When we ran it three times, we obtained the values 6715, 8047, and 9576. Your own numbers will probably differ. The lesson here is that your programs should not make any assumptions concerning the loop control value.

One final note on syntax: a frequent source of confusion to beginning Pascal programmers is the seeming ambiguity of the term *statement.* Consider the **for** statement we have been using:

```
for i := initial downto final do
 begin
  writeln(i : 11 - i);
  final := 5;
  sysbeep(45)
 end;
```

We have been using the word *statement* to apply to (1) the entire **for** statement extending from the word **for** to the word **end**, (2) the compound statement that starts at **begin** and ends at **end**, and (3) each of the three lines of the program that lie betweenthe **begin** and the **end**. You might object to calling all these things statements; surely, you think, we must be getting sloppy with our terminology.

In fact, we aren't. Pascal simply allows statements to contain other statements, which may, in turn, contain statements as well. In the for_lab program, for example, the **for** statement contains a compound statement, which itself contains three single-line statements. In general, Pascal allows you to "nest" statements to an arbitrary depth: there is no restriction in the language as to what depth statements may be nested within each other. As your experience with Pascal grows, you'll come to appreciate the elegance of this design.

THE REAL TYPE

We said earlier in this chapter that integers were numbers without decimal points. In mathematics, numbers with decimal points are called *real numbers*. Pascal provides the real type to represent mathematical real numbers.

Clear any program you currently have in memory, saving it first if you wish. Then enter the following program:

```
program real_lab;
{ Experiments on reals }

var
 x, y : real;

begin { real_lab }
 x := 1.2;
 y := 3.4;
 writeln('x is ', x);
 writeln('y is ', y);
 writeln;
 writeln('x + y is ', x + y);
 writeln('x - y is ', x - y);
 writeln('x * y is ', x * y);
 writeln('x / y is ', x / y)
end.
```

The program, when run, should give the following output:

```
x is  1.2e+0
y is  3.4e+0

x + y is  4.6e+0
x - y is -2.2e+0
x * y is  4.1e+0
x / y is  3.5e-1
```

Most of the arithmetic operations on real number variables are performed with the same symbols that worked on integer variables: a plus sign for addition, a minus for subtraction, and an asterisk for multiplication. The slash, which you may remember didn't work well for dividing integer variables, is the symbol to use when you want to divide two real values. However, the **div** and **mod** operations won't work on real values.

The default formatting for printing real values is *scientific notation* (also called *floating-point notation* or, most simply, *e notation.*) The letter e in a number expressed in scientific notation should be read "times ten to the power of." To figure the value of a number expressed in floating-point notation, multiply the number in front of the e (the mantissa) by the power of 10 following the e (the exponent). Hence, in our output, 4.6e+0 translates as 4.6×10^0, or just 4.6. (In case you have forgotten some of your math, $10^0 = 1$.)

The default formatting for real values also specifies a default field width of ten spaces. Pascal allows one space for a possible sign on the front of the number and five spaces for the exponent (including its sign). The e takes up a space too; all this leaves three spaces for the mantissa. These three spaces are taken up by the leading digit, a decimal point, and one digit following the decimal.

Because the default field width only allows one digit to be displayed past the decimal point, the multiplication and division results shown in our example were rounded to tenths. What if we want more precise answers? The answer is to modify the field width directly in the program, just as we did with integer values. Try changing the writeln statements in the program to the following:

```
writeln('x is ', x : 15);
writeln('y is ', y : 15);
writeln;
writeln('x + y is ', x + y : 15);
writeln('x - y is ', x - y : 15);
writeln('x * y is ', x * y : 15);
writeln('x / y is ', x / y : 15)
```

Expanding the field width to 15 allows six digits to be printed after the decimal point. Now the output looks like this:

```
x is  1.200000e+0
y is  3.400000e+0

x + y is  4.600000e+0
x - y is -2.200000e+0
x * y is  4.080000e+0
x / y is  3.529412e-1
```

Floating-point notation is ideal for printing values that are very small or very large, but you'll often want to display real values in their normal form. This is known as *fixed-point notation.* To signal Pascal that it is to print out a real value in fixed-point notation, you add a second colon after the field width specification followed by a second number specifying the number of digits you want printed following the decimal point. To see how this works, try modifying the real_lab program once more:

```
writeln('x is ', x : 6 : 3);
writeln('y is ', y : 6 : 3);
writeln;
writeln('x + y is ', x + y : 6 : 3);
writeln('x - y is ', x - y : 6 : 3);
writeln('x * y is ', x * y : 6 : 3);
writeln('x / y is ', x / y : 6 : 3)
```

This time we've asked that the output be printed in fixed-point format allowing three digits past the decimal point:

```
x is  1.200
y is  3.400

x + y is  4.600
x - y is -2.200
x * y is  4.080
x / y is  0.353
```

You should feel free to experiment with different combinations of field width and specifications for the number of digits after the decimal.

The rules for writing real numbers in Pascal are a little more complex than those for writing integer values:

- Do not use commas in numbers; write 1024.137, not 1,024.137.
- You may write real numbers with no fractional part as integer values. The general rule is that if Pascal sees an integer where it expects a real number, the integer is automatically converted into a real number.
- A number must have a digit on both sides of the decimal point; write 2.0 and 0.314, not 2 or .314.

- A negative real value is written with a leading minus sign: −2.43, for example.
- You may write real constants using scientific notation. Write the mantissa as you would a fixed-point number. Follow it with an e (or an E) and then an integer representing the power of 10 by which you want to multiply the mantissa (with an optional sign).

As usual, you might want to verify that these rules work and see what happens when you break them.

As is true of integers, there are an infinite number of mathematical real numbers, and a computer with a finite amount of memory can only represent a finite number of different values. As a result, the real type in Pascal is limited to the range of the real numbers it can represent.

To discover these limits, enter the following program either by modifying the real_lab program or by clearing it and starting over:

```
program real_lab;
{ Experiments on reals }

 const
  START = 1.0;
  MULTIPLIER = 2.0;
  MAXTIMES = 200;
  WIDTH = 16;

 var
  x : real;
  i : integer;

begin { real_lab }
 x := START;
 for i := 1 to MAXTIMES do
  begin
   writeln(x : WIDTH);
   x := MULTIPLIER * x
  end
end.
```

You'll notice that this program uses something we haven't seen before in our programs: between the program line and the variable definition part there is a constant definition part. The syntax sketch for a constant definition part is shown in

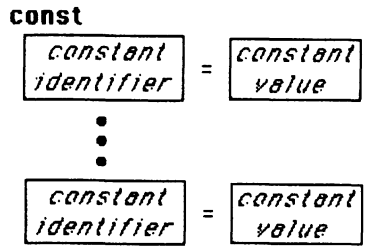


Figure 2-6.

Constant definition part syntax

Figure 2-6. Figure 2-7 shows where the constant definition part fits within a Pascal program.

In general, the constant values listed on the right side of the equal signs in constant definitions may be

- Numeric constants of either real or integer types.
- A previously defined constant identifier (with an optional sign if the constant identifier represents a number).
- A string.

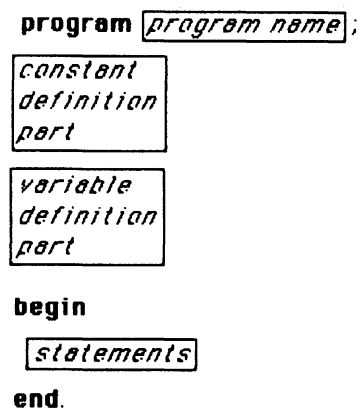


Figure 2-7.

Pascal program syntax
(still incomplete)

This is the legal syntax for a constant definition part; now, what does it do? Simply speaking, Pascal "remembers" both the identifier and the value in each definition. When it comes across a constant identifier later in the program, it acts just as if the corresponding constant value were in that spot in the program instead.

Given our constant definitions, the program here will be executed as if the following had been written for the executable part:

```
begin { real_lab }
 x := 1.0;
 for i := 1 to 200 do
  begin
   writeln(x : 16);
   x := 2.0 * x
  end
end.
```

Why go to the bother of writing constant definitions? Why not just use the actual numbers or strings in the program itself? Often a constant identifier is more meaningful to someone reading the program than a bare constant value would be. The word MULTIPLIER, for example, suggests the purpose of the constant, while the number 2.0 all by itself might not.

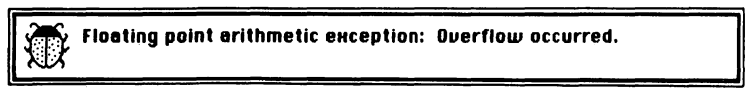
Constant identifiers can make writing a large program much easier. For example, if you always want to print numeric values in a certain field width, it will be easier to remember to write the constant identifier WIDTH in the program instead of remembering the exact value of the field width you decided on each time you need to specify it.

Finally, assume you later want to change the field width. When you have defined the field width as a constant, all you need to do is change the single constant definition, which is much easier than going through the program and changing every occurrence of the field width.

Although the same rules apply to naming constant identifiers as apply to variable identifiers, constants are *not* variables. By definition they cannot change in value while the program is being run. It is an error, for example, to put a constant identifier on the left side of an assignment statement. We write all our constant identifiers in uppercase and all our variables in lowercase to emphasize the distinction between the two.

Let's look at the program that sparked this digression. The real variable x is initialized to the value START. Then the following two things happen MAXTIMES times: the value of x is printed, and then x is multiplied by MULTIPLIER. Since the value of MULTIPLIER is 2.0, this means that x will increase rather quickly; run the program to find out just how quickly.

When more than one hundred lines are displayed, the program crashes. The bug box displays the error message:



The term *overflow* means that a variable's value has become too large to be held in the storage reserved for the variable. (*Exception* is used here to describe an unusual situation.) Examining the program's output will tell you that the variable x last held the value 1.7014118×10^{38}(this was the final value printed). However, multiplying this value by 2 caused overflow. This means that the biggest number a real variable can represent is somewhere between 1.7014118×10^{38} and twice that number.

If you want a better estimate on the largest real, try changing the constants START and MULTIPLIER as follows:

```
START = 1.7e38;
MULTIPLIER = 1.01;
```

Then rerun the program. Play with different values of START and MULTIPLIER, if you wish, to come as close as you can to the largest real value. The answer, in fact, is approximately 3.4×10^{38}. Of course, you may also have negative numbers this large.

Limitations on memory also prevent Pascal from representing real numbers very close to zero. To see how this works, change the program slightly to make the numbers get smaller instead of bigger:

```
START = 1.0;
MULTIPLIER = 0.5;
MAXTIMES = 152;
```

This time, there is no fatal program error, but you'll note the numbers printed out suddenly go from non-zero values to zero values. This is known as *underflow*, which is another way of saying that a variable's value came too close to zero to be represented. You may want to play with the numbers again to get a good estimate on the smallest possible positive real value. You should obtain, roughly, 1.4×10^{-45}.

You have learned that you cannot represent very big or very tiny mathematical real numbers using Pascal's real type. Pascal is also limited in the amount of precision with which it can represent real values. In mathematics there are not only an infinite number of real numbers in total, but there are also an infinite number of real numbers between any two real numbers. But Pascal can only represent a finite number of different real values. In other words, there are many real numbers that Pascal cannot represent exactly (an infinite number of them, in fact).

To demonstrate this point, try the following program:

```
program real_lab;
{ Experiments on reals }

 const
  ADDITIONS = 10;
  WIDTH = 16;

 var
  i : integer;
  reciprocal, sum : real;

begin { real_lab }
 reciprocal := 1.0 / ADDITIONS;
 sum := 0.0;
 for i := 1 to ADDITIONS do
  sum := sum + reciprocal;
 writeln('The value ', reciprocal : WIDTH);
 writeln('added together ', ADDITIONS : 1, ' times');
 writeln('gives the sum ', sum : WIDTH);
 writeln('which differs from the exact');
 writeln('answer (1.0) by ', sum - 1.0 : WIDTH);
end.
```

The program gives instructions to add 1/10 + 1/10 + ... + 1/10, where there are ten 1/10's in all. The answer, in theory,

should be exactly 1. We initialize sum, a variable to hold the result, to 0 and then perform the calculation in a **for** loop that is executed 10 times. Each time through the loop, we add the value of the reciprocal, which was initialized to 1/10, to sum. On exit from the loop, we print out the result and also the difference between the result and the mathematical result. The printout should appear as follows:

```
The value  1.0000000e-1
added together 10 times
gives the sum  1.0000001e+0
which differs from the exact
answer (1.0) by  1.1920929e-7
```

Pascal's result differs by about one part in ten million from the "exact" answer. This is a very good performance for all practical purposes. The lesson you should extract from all this is that you should never rely on the result of a real calculation to give you an exact answer.

THE BOOLEAN TYPE

So far you have met two of Macintosh Pascal's numeric data types, integer and real. Pascal also has non-numeric data types, and one of the most important is the Boolean type, which we'll explore in this section. The Boolean type is named for George Boole (1815-1864), a mathematician who developed the theory of binary logic used by nearly all computers.

There are only two possible Boolean values, TRUE and FALSE. We have chosen to write them in uppercase to emphasize the fact that they are constants of the Boolean type, although you may use lowercase letters or capitalize them, whatever you prefer. Since Booleans can have only two possible values, they are useful in controlling how your programs are executed, as we will see.

Let's begin with a simple program using Boolean variables.

```
program Boo_lab;
{ Experiments with Booleans }

var
 b1, b2 : Boolean;
```

```
begin { Boo_lab }
 b1 := TRUE;
 b2 := FALSE;
 writeln(b1);
 writeln(b2);
end.
```

Try typing in this program. It simply assigns values to the two Boolean variables b1 and b2 and then prints out those values. The result is simply

```
True
False
```

This should show you that there is nothing especially mysterious about Boolean variables. They can take on values (just like any other variable) and their values can be displayed using the writeln statement (just like any other variable). It is generally a mistake, however, to mix different types of variables. See what happens when you replace the line

```
b1 := TRUE
```

with the line

```
b1 := 10.3
```

When you try to run this program, the result is the "An incompatibility between types has been found" bug box you have seen before. Pascal is relatively picky about not allowing you to put a value of one type into a variable of another type. This is to protect your variables from taking on unexpected values; you may rely on real variables having only real values, Boolean variables having only the values TRUE or FALSE, and so on.

As with all the items written by the writeln command, you may specify a field width for a Boolean. To see how it works, try modifying the program as follows (remember to fix the bug first):

```
program Boo_lab;
{ Experiments with Booleans }

 var
```

```
  b : Boolean;
  i : integer;

begin { Boo_lab }
 b := TRUE;
 for i := 1 to 10 do
  writeln(b : i);
end.
```

When you run the program, it looks like this:

```
T
Tr
Tru
True
 True
  True
   True
    True
     True
      True
```

Next try modifying the program to print "False" instead of "True" and then run it to observe the results. It appears that the rules for "formatted" Boolean output are as follows: If the specified field width is less than that needed to print the word "True" or "False," the word is truncated to the specified length. If the field width is greater than the length of "True" or "False," the word is padded with blanks on the left to expand it the appropriate width.

It doesn't make any sense as far as Pascal is concerned to perform any of the arithmetic operations we've already seen on Boolean values. Try the following program, which attempts to add two Booleans:

```
program Boo_lab;
{ Experiments with Booleans }

 var
  b1, b2, b3 : Boolean;

begin { Boo_lab }
 b1 := FALSE;
```

```
  b2 := TRUE;
  b3 := b1 + b2;
  writeln(b3);
end.
```

You'll get the same outcome no matter what arithmetic operation you try: the "type incompatibility" bug box. Instead, Pascal provides three operators especially for Boolean values. The operators are represented by the reserved words **and**, **or**, and **not**.

The simplest Boolean operator is **not**; enter the following to see how it works.

```
program Boo_lab;
{ Experiments with Booleans }

var
 b1 : Boolean;

begin { Boo_lab }
 writeln('b1' : 7, 'not b1' : 13);
 writeln('--' : 7, '------' : 13);
 for b1 := FALSE to TRUE do
  writeln(b1 : 7, not b1 : 13)
end.
```

The result is

```
     b1        not b1
     --        ------
  False          True
   True         False
```

The **not** operator applies to a single Boolean value and turns it around; TRUE is turned into FALSE, FALSE into TRUE. In the jargon, **not** is called a *unary* operator because it only acts on one value. On the other hand, the arithmetic operators (+, −, *, /, **div**, and **mod**) are called *binary* operators because they work on the two values found on either side of the operator.

Note, by the way, that it is perfectly all right to have a **for** loop with a Boolean loop control variable. As far as Pascal is concerned, the value of FALSE is "less than" the value of TRUE, so the loop is done twice: once with the loop control variable set to FALSE, the second time to TRUE. (A quick

review question: What would happen if you swapped the initial and final values, TRUE and FALSE, in the **for** statement? Try it. Then change it back.)

The **and** operator is a binary Boolean operator that works on two values. To see how it works, modify the program to look like this:

```
program Boo_lab;
{ Experiments with Booleans }

 var
  b1, b2 : Boolean;

begin { Boo_lab }
 writeln('b1' : 7, 'b2' : 7, 'b1 and b2' : 13);
 writeln('--' : 7, '--' : 7, '---------' : 13);
 for b1 := FALSE to TRUE do
  for b2 := FALSE to TRUE do
   writeln(b1 : 7, b2 : 7, b1 and b2 : 13)
end.
```

You may think it a little strange to see one **for** statement immediately after another. But this is just another example of nesting Pascal statements within one another. As far as the first **for** statement is concerned, the "statement" that it is to execute repetitively is the second **for**. The second **for**, in turn, executes the writeln statement repetitively. When you run the program, this should be the result:

```
    b1      b2    b1 and b2
    --      --    ---------
 False   False        False
 False    True        False
  True   False        False
  True    True         True
```

Note that the values of the loop control variables take on all four possible combinations of TRUE and FALSE. The output shows that when two Boolean values are combined with the **and** operator, the result is TRUE only if both values are TRUE. All other combinations give the result FALSE. (If you don't quite understand why this program prints four lines, reread the comments on statement nesting and the **for** loop.)

The third Boolean operator, **or**, is also a binary operator. Change the program, replacing **and** everywhere by **or**:

```
program Boo_lab;
{ Experiments with Booleans }

 var
  b1, b2 : Boolean;

begin { Boo_lab }
 writeln('b1' : 7, 'b2' : 7, 'b1 or b2' : 13);
 writeln('--' : 7, '--' : 7, '--------' : 13);
 for b1 := FALSE to TRUE do
  for b2 := FALSE to TRUE do
   writeln(b1 : 7, b2 : 7, b1 or b2 : 13)
end.
```

Running the program shows the difference **or** makes:

```
     b1      b2     b1 or b2
     --      --     --------
  False   False        False
  False    True         True
   True   False         True
   True    True         True
```

Combining two Boolean values with the **or** operator gives the result FALSE only if both Booleans are themselves FALSE; otherwise it gives the value TRUE. Like the **and** operator, **or** mirrors its usual dictionary meaning.

So far we have discussed arithmetic operators that work on numeric values and give numeric results, and Boolean operators that work on Boolean values and give Boolean results. The final class of operators are called the *relational* operators. Relational operators generally work on numeric values and give Boolean results. The six relational operators used in Pascal are shown in Table 2-1. They are all binary operators, and it's easiest to think of them as comparing two values and giving the result of the comparison.

To see how the relational operators work, it's best to try them out in a program. Enter the following:

```
program Boo_lab;
{ Experiments with Booleans }
```

```
var
 i, j : integer;

begin { Boo_lab }
 i := 12;
 j := 13;
 writeln('i is ', i : 1);
 writeln('j is ', j : 1);
 writeln('i = j is ', i = j);
 writeln('i <> j is ', i <> j);
 writeln('i < j is ', i < j);
 writeln('i > j is ', i > j);
 writeln('i <= j is ', i <= j);
 writeln('i >= j is ', i >= j)
end.
```

This sets the values of two numeric variables and displays the Boolean values obtained by comparing the numbers with each of the six relational operators. For the values shown, you should get these results:

```
i is 12
j is 13
i = j is False
i <> j is True
i < j is True
i > j is False
i <= j is True
i >= j is False
```

Table 2-1.

Pascal Relational Operators

Operator	Meaning
=	equal to
<>	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

For example, the expression "i <= j" has the value TRUE because the value of i is less than or equal to the value of j; otherwise it would take on the value FALSE.

You could experiment with the relational operators by changing the constants assigned to i and j in the first two assignment statements of the program. Up to now, whenever we have wanted a variable set to a value in a program, we have used the assignment statement. If we wanted to set a variable to a different value in a subsequent run of the program, we had to change the assignment statement so that the desired value was on the right side of the assignment symbol (:=).

We can do all this in another way by using a slightly more flexible method of assigning constant values to variables. Change these lines:

```
i := 12;
j := 13
```

to the following:

```
writeln('Enter a value for i:');
readln(i);
writeln('Enter a value for j:');
readln(j)
```

This introduces a new Pascal command, *readln.* Like the assignment statement, readln sets the value of a variable but instead of plugging that value into the program itself, the value is typed at the Macintosh keyboard when the program is run.

Now run the program. The first thing you'll see is the prompt generated by the first write statement:

```
Enter a value for i:
```

The program waits for you to type something in at the keyboard. Type 12 and press the RETURN key. The second prompt for a value for j is then displayed; type 13 and press RETURN again. You should see the same results as you had before.

Run the program again, entering different numbers for i and j. If you make a typing mistake before you press RETURN, use the BACKSPACE key to correct it. In addition to experimenting with the relational operators, this is also a good way

to gain experience with the way the readln command works. Run the program a number of times and answer the following questions:

- What happens if, instead of a number, you try to type "illegal" characters not allowed in an integer constant (letters, for example)?
- Can you type blanks before you type the number? Can you type blanks after you type the number? Can the number contain embedded blanks? Can you type illegal characters if they are separated from the number by trailing blanks? What happens if they aren't separated by blanks?
- What happens if you try to enter a number larger than 32767 or smaller than −32767?

Other experiments may occur to you as well. After you have learned the rules for entering integer values, try the same thing with real values. The simplest way to do this is to change the variable declaration from integer to real. To avoid confusion, also change the prompts to ask for reals instead of integers. Make sure you can enter real values in both fixed-point and floating-point notation. What happens if you try to break the rules for writing real constants discussed earlier? We will continue our discussion of the readln statement in Chapter 6.

A final note on the relational operators: we have shown them working with numeric values, but you can also use them to compare Boolean values as well. (Remember, FALSE is considered "less than" TRUE.) As we go on to explore more Pascal data types, we'll see that most of them can be compared with the relational operators as well.

THE while LOOP

Earlier in the chapter, we explored the **for** loop and saw how useful it was for executing statements a specified number of times. Often, however, you will not be able to specify exactly how many times a loop should be executed at the time you write the program. Instead, execution of the statements inside the loop will depend on a condition that must be satisfied first. If the condition is satisfied, the statements are executed and the condition is checked again. If the condition is still satisfied, the statements are executed again; then the

condition is checked again, and so on. If it ever happens that the condition is not satisfied, the loop exits; program execution continues with the next statement following the loop.

For example, you might use the following method to find something good to watch on television: turn on the set; if you don't like what you see, change the channel. If you don't like that, change the channel again. Keep going until you find something you do like; then sit down and watch that. To express this in a Pascal-like notation, you might write

```
turn on the TV

while you don't like what is on:
   change the channel

relax and enjoy
```

Here the *loop body* (the action performed repetitively) is the statement "change the channel." The *loop condition* is that you dislike what's on; as long as the condition (that you don't like what is on) is met, the loop body is executed. When the condition is not satisfied—you finally find something interesting—you exit from the loop and proceed with the next statement: relax and enjoy.

Assume the only interesting show is on channel 5, for example. To simulate the channel-changing process in true Pascal, one could write the following program:

```
program while_lab;
{ Experiments with while loops }

 const
  INTERESTING = 5;

 var
  ch : integer;

begin { while_lab }
 ch := 2;
 writeln('Turning on TV...');
 while ch <> INTERESTING do
  begin
   write('Channel ', ch : 1, ' is boring..');
   writeln('Click!');
   ch := ch + 1
```

```
  end;
  writeln('Ah, Channel ', ch : 1, ' has something interesting
  writeln('Please pass the popcorn.')
end.
```

Note that the loop condition is expressed with a relational operator: "ch<>INTERESTING." As we discussed in the previous section, comparing two numbers using a relational operator gives a Boolean result, either TRUE or FALSE. If the condition is satisfied (the channel is uninteresting), the comparison will give the result TRUE, and the loop body is executed. (Here, the loop body is the compound statement following the word **do**.) Once the loop condition is not satisfied (the channel is interesting), the comparison gives a FALSE, and control is passed to the next statement following the **while** statement.

Type in and run the previous program. The following should be the output:

```
Turning on TV...
Channel 2 is boring..Click!
Channel 3 is boring..Click!
Channel 4 is boring..Click!
Ah, Channel 5 has something interesting.
Please pass the popcorn.
```

The syntax of the **while** statement is shown in Figure 2-8. Note that the loop body is always a single statement. (Of course, the "single statement" can be a compound statement, as it was in our program.) The loop condition is always a Boolean expression; it must evaluate to either TRUE or FALSE. As long as it evaluates to TRUE, the loop body is executed. Once it becomes FALSE, the program exits from the loop and executes the first line following the loop.

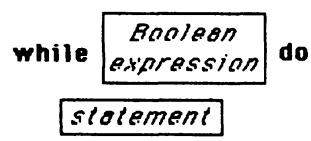


Figure 2-8.

while statement syntax

It is also worth emphasizing that the **while** statement is a statement and it can go anywhere any other kind of statement we have discussed can go. And any kind of statement can become the loop body of a **while** statement.

Like the **for** statement, it is possible for the loop body of the **while** statement not to be executed at all. To see this, change the initial value of the variable ch from 2 to 5. The output this time will not show any channel-changing at all:

```
Turning on TV...
Ah, Channel 5 has something interesting.
Please pass the popcorn.
```

Unlike the **for** statement, it is possible for the **while** statement to execute "forever." To see this behavior, try changing the initial value of the variable ch from 5 to 7 and then running the program. Note that the program shows no signs of stopping, even after it exhausts all the channels in both VHF and UHF bands.

If you waited long enough, of course, this program would end with an execution error after it had checked Channel 32767 and found it dull. (Do you remember why?) If you don't want to wait that long, use the mouse to move the pointer to the word Pause in the menu bar. The Pause option is available only while a program is running. Press the mouse button to cause the program to stop temporarily; release it to resume program execution. (Note that a pointing finger appears in the Program window when you pause the program. This shows what statement in the program will be executed next. In Chapter 5 we'll discuss how to use the pointing finger.) To stop your program, choose the Halt option from the Pause menu.

As another example, let's write a program to calculate the greatest common divisor of two integers. The greatest common divisor (or GCD, for short) of two integers is defined as the largest integer that divides the two integers evenly. For example, the GCD of 51 and 34 is 17, and the GCD of 81 and 128 is 1.

One method used to calculate GCDs is called *Euclid's algorithm.* To calculate the GCD of two positive integers, m and n, the algorithm involves the following steps:

1. Let remain be the remainder when you divide m by n.
2. If remain is 0, the GCD is n; the algorithm is done.

3. If remain is non-zero, set m to the value of n, set n to the value of remain, and then return to step 1.

You should verify that you understand the steps of the algorithm. For example, to find the GCD of 100 and 80,

- After step 1, m is 100, n is 80, and remain is 20.
- Skip step 2, since remain is not 0.
- After step 3, m is 80 and n is 20.
- After step 1 again, remain is 0.
- Thus, step 2 tells you that the GCD is 20.

If you remember that the **mod** operator gives the remainder when two integers are divided, Euclid's algorithm translates easily into Pascal.

The difference between the algorithm and the program is that the algorithm calculates the remainder both above the loop and also inside the loop. This is important, since you need to know the first value of remain before you start the loop. Try typing in and running this program as follows, and verify that it gives correct results no matter what two positive integers you enter. (What happens if one or both numbers are not positive?)

```
program Euclid;
{ Calculate the GCD of two positive integers }

 var
  m, n, remain : integer;

begin { Euclid }
 write('Enter the first integer:');
 readln(m);
 write('Enter the second integer:');
 readln(n);
 write('The GCD of ', m : 1, ' and ', n : 1, ' is ');
 remain := m mod n;
 while remain <> 0 do
  begin
   m := n;
   n := remain;
   remain := m mod n
  end;
 writeln(n : 1)
end.
```

THE repeat LOOP

The third and last loop structure in Pascal is the **repeat** loop. The basic idea is the same as the **while** and **for** loops: one or more statements (the loop body) are executed over and over; the looping stops when a specified condition is met. The syntax of the **repeat** loop is shown in Figure 2-9. The statements between the **repeat** and **until** are executed over and over until the condition of the Boolean expression becomes TRUE. Note that no compound statement is necessary if you want to put more than one statement in the loop body; simply separate one statement from the next with semicolons as always.

The **repeat** loop is useful when the loop body is to be executed at least once. Suppose you wanted to write a simple desk calculator program that would accept any number of input numbers and print their sum. The main problem is letting the program know that the person running the program has finished entering numbers; let's say this will be done by entering a 0 value.

The strategy of this program might be expressed using an informal notation halfway between Pascal and English:

```
initialize sum to zero
repeat
   get a number
   add number to sum
until number is zero
print sum
```

Such an informal program description is often called *pseudo-code*. Because pseudo-code has no hard and fast rules, it is often easier to design a program roughly in pseudo-code before you actually write it in formal Pascal. We'll make occasional use of pseudo-code in the remainder of the book.

Translating the pseudo-code into Pascal gives the following program:

```
program desk_calc;
{ Add numbers until a zero is entered }

 var
  sum, x : real;

begin { desk_calc }
 sum := 0.0;
```

```
repeat
 write('Enter a number (0 to end):');
 readln(x);
 sum := sum + x
until x = 0.0;
writeln('The sum is: ', sum : 16)
end.
```

Type in and run the program; make sure you can add up any sequence of numbers that occur to you. Note that this program could also have been written using a **while** loop, as shown here:

```
program desk_calc;
{ Add numbers until a zero is entered }

var
 sum, x : real;

begin { desk_calc }
 sum := 0.0;
 write('Enter a number (0 to end):');
 readln(x);
 while x <> 0.0 do
  begin
   sum := sum + x;
   write('Enter a number (0 to end):');
   readln(x)
  end;
 writeln('The sum is: ', sum : 16)
end.
```

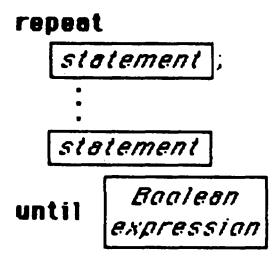


Figure 2-9.

repeat statement syntax

Both programs do exactly the same thing; the only difference is that, since the readln must be done at least once, the version of the program using **repeat** is slightly shorter.

As another example of the use of **repeat**, let's design a program that calculates the square root of an input real number. Many programming languages provide a built-in square root routine that can be easily used with any program. Pascal provides a built-in square root as well, but for the purposes of this program let's assume that it is unavailable. (We'll see how it works in Chapter 7.)

A remarkably easy and quick way to calculate square roots on the computer is to use *Newton's method.* Assume the number we are trying to get the square root of is x. Also assume we have made an estimate of the value of the square root; call this guess g. According to Newton's method, a better estimate for the square root is given by this formula:

$$\text{betterg} = \frac{1}{2}\left(g + \frac{x}{g}\right)$$

One strategy for calculating the square root would be to repeat this calculation over and over, getting closer and closer estimates of the square root. The only question is the loop condition: when should the program exit from the loop? One possibility, and probably the easiest one to write, is to exit from the loop when two successive estimates of the square root are equal.

In pseudo-code, the strategy for our program might be written as follows:

```
get a value for x
initialize new_guess
repeat
   set old_guess to new_guess
   get new_guess from formula
      (using value of old_guess)
until old_guess and new_guess are equal
print new_guess as approximate square root
```

Here old_guess and new_guess are two successive approximations of the square root of x. The method must start with an initial guess of the square root; the program should initialize the value of new_guess with this initial guess before the

repeat loop is entered. (Why?) The first guess doesn't have to be a very good guess; you can initialize new_guess to x/2 in the Pascal program:

```
program Newton;
{ Calculate square roots by Newton's method }

var
 x, old_guess, new_guess : real;

begin { Newton }
 write('Enter a number:');
 readln(x);
 new_guess := x / 2;
 repeat
  old_guess := new_guess;
  new_guess := (old_guess + x / old_guess) / 2
 until new_guess = old_guess;
 writeln('The square root of ', x : 16, ' is ', new_guess : 16)
end.
```

Type in and run this program; enter a value of 2 for x when requested. The program should produce the answer 1.4142135e+0, which is a good estimate of the correct value. Try a few other values and see if the results are equally as good.

The key line in this program is the calculation of new_guess based on the value of x and old_guess inside the **repeat** loop. The expression that accomplishes the calculation is a more complex one than we have discussed previously, involving parentheses, addition, and two divisions. We will return to this expression in the following section and explain it fully.

You may note that this program doesn't work well if you enter a zero value or a negative value. A zero entry causes division by 0, which is illegal. If a negative value is entered, the program goes into a never-ending loop; successive estimates never converge into a single number. (If you enter a negative number, you'll have to stop the program by choosing Halt from the Pause menu.)

One solution to these problems is to have the program check its input for these situations before it attempts to do its calculations; we'll discuss the subject of input checking further in Chapter 4.

EXPRESSIONS: PRECEDENCE AND PARENTHESES

All through this chapter, you have been using expressions in a number of different places in your programs. You have seen expressions used in the following ways:

- In an assignment statement, there must be an expression on the right side of the := symbol.
- There must be Boolean expressions between the words **while** and **do** in a **while** statement and after the word **until** in a **repeat** statement.
- The initial and final values in a **for** statement are both expressions.
- Finally, the values printed by write and writeln statements are, in general, specified by expressions as well. The field width and "digits after the decimal" formatting specifications used by write and writeln are also expressions.

It's easy to see, therefore, that expressions pervade nearly every Pascal program you write. Just as it was important that you get a grasp on the concept of a Pascal statement, it is important that you understand how Pascal expressions work. That is our goal in this section.

There are two important facts to remember about expressions used in a Pascal program. First, when a statement containing an expression is executed in a program, the expression is *evaluated;* that is, the computer goes through whatever operations are necessary to figure out the value represented by the expression. Second, any expression in a Pascal program can be considered to be of a definite *type,* just as variables are of definite types. So when Pascal wizards talk about Boolean expressions, they actually mean expressions that evaluate to a Boolean value, TRUE or FALSE.

For the following discussion, assume you have defined variables in a program as follows:

```
var
  x, y, z : real;
  i, j, k : integer;
  b1, b2, b3 : boolean;
```

The simplest kind of expressions are the constant values. For

example, consider these assignment statements:

```
x := 3.25;
i := 5280;
b3 := TRUE
```

Since anything that appears on the right side of an assignment is an expression by definition, the constants 3.25, 5280, and TRUE are all full-fledged expressions, albeit simple ones. All three also have definite types: real, integer, and Boolean.

You have also seen that variable names can be considered expressions. For example,

```
z := x;
j := i;
b1 := b2
```

Here x, i, and b2 are all expressions of real, integer, and Boolean types, respectively.

Next, you have also seen that operators can be used in expressions to get the computer to perform arithmetic, Boolean, and comparison operations. In these examples,

```
y := x * z;
k := i - j;
b3 := b1 or b2
```

once again the expressions are evaluated and the resulting values are assigned to the variables.

With only one exception, Pascal does not allow types to be mixed; it is against the rules of the language to use one type where Pascal expects some other type. This is Pascal's rule against *type mixing.* We have tried to break this rule a number of times in our experiments, and the result was usually the "incompatibility between types" bug box. For example, you know from experience that it is illegal to assign a real expression to a Boolean variable, or vice versa.

The single exception to this rule is simple and reasonable: you may use an integer expression where Pascal expects a real expression. Thus, the following assignments are all legal even though they involve type mixing:

```
x := 3;
y := i;
z := j div k
```

In all these cases, the integer expressions are evaluated and then converted to real values. Those real values are the ones actually assigned to the real variables. This is just a special case of a more general rule: whenever integer constants, variables, or expressions are combined with real constants, variables, or expressions, the integers are converted into reals before the specified operations are performed. So in the following assignments, the integer values act just as if they were real values:

```
x := 3 + y;
y := z - i;
z := k * x
```

Remember that this integer-to-real conversion rule is an exception to the general rule against type mixing in Pascal. Pascal allows you a little leeway in distinguishing between reals and integers, but you will almost certainly run into problems if you try mixing any other types.

The final rule about type mixing is one you brushed up against earlier in this chapter: the / real division operator will always give a real result, even if it is used to divide two integers. This is in contrast to the addition, subtraction, and multiplication operators. When used on two integers, these operators will always give an integer as a result.

With Pascal's rules on type mixing understood, it becomes possible to tackle more complex expressions. Pascal allows you to combine an arbitrary number of operations into a single expression; for example:

```
b1 := b1 and b2 or b3;
x := 3.0 * x + z / y;
i := i - j - k mod 10
```

All these expressions are still relatively simple, but it is easy to write more complex combinations. A problem arises when one tries to predict in what order the operations specified in a complex expression will occur. The Macintosh can only do one thing at a time, and if you stop and think a bit, you'll see that the order in which operations are performed will affect the final value of the expression. For example, consider the assignment

```
x := 2.0 * 3.6 + 0.4
```

If Pascal performs the multiplication first, x will take on the value 7.6. If the addition is done first, x will have the value 8.0.

To clear up such ambiguities, Pascal provides *precedence rules* (rankings) that govern the order in which operations are performed in any expression. One way—and probably the simplest way—to understand the precedence rules is to think of the Pascal operators as having high or low precedence. In an expression that contains a number of operations, the operations of highest precedence are done first, the operations of next highest precedence are done next, and so on. Operations of lowest precedence are performed last.

Figure 2-10 shows the precedence of the Pascal operators we have seen so far. The highest precedence operator is **not**; the operators *, /, **div**, **mod**, and **and** are next highest, followed by the operators +, −, and **or**. The lowest precedence operators are the six relational operators: =, <, >, <=, >=, and <>. Looking at the previous example, since multiplication has precedence over addition and is performed first, x takes on the value 7.6.

If you find it necessary to alter the natural precedence of operations in a Pascal expression (and you often will), you may do so with parentheses. Operations specifed within parentheses are always done first when the expression is evaluated, regardless of the precedence of the operators involved.

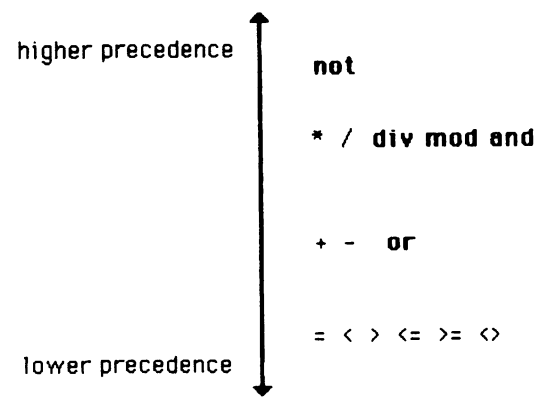


Figure 2-10.

Pascal operator precedence

In effect, you should think of parentheses in an expression as containing *subexpressions* whose values must be calculated first; those values are then used to evaluate the entire expression. The subexpressions may themselves contain parentheses, and the same rule applies to these sub-subexpressions: operations within parentheses are done first. Parentheses may be "nested" to any depth within an expression.

Let's look again at the key line from the Newton program in the previous section, which will demonstrate nearly everything discussed so far in this section:

```
new_guess := (old_guess + x / old_guess) / 2
```

Here the subexpression in parentheses is evaluated and the value obtained is divided by 2. (Remember that it would be equally correct to write "2.0" instead of "2"; the division is a real division in either case.) In evaluating the subexpression within parentheses, Pascal does the division x/old_guess first because the / operator has higher precedence than the + operator. The result of the division is then added to the value of old_guess.

Understandably, you may not remember which operators have precedence when writing a program, but you can always use "insurance" parentheses to guarantee Pascal will do things in the order you want. For example, the calculation just discussed could have also been written as

```
new_guess := (old_guess + (x / old_guess)) / 2
```

Here the parentheses around the division "x/old_guess" are absolutely useless as far as Pascal is concerned. They do perform an important function, however; they may make the order of operations clearer to the person writing or reading the program. Until you are comfortable with the Pascal precedence rules, use insurance parentheses to remove any worries you have about precedence.

At times you may find yourself slightly confused about the binary operators + and − (used for addition and subtraction) and the + and − used to give an algebraic sign to a numeric quantity. When used as signs, + and − are considered to be unary operators; they operate on the single value following them. Their precedence, however, remains the same whether they are used as binary or unary operators.

In an expression involving operators of equal precedence,

the operations are performed from left to right. For example, the assignment statement

```
i := 9 - 6 - 1
```

assigns the value 2 to x (not 4) because 9–6 is performed first.

Even though the rules Pascal uses in evaluating expressions are relatively simple, it is very easy for both beginning and advanced programmers to fail to think through all the implications of the rules and, as a result, come up with syntax errors or erroneous results in their programs.

Suppose you wanted to write a loop statement that would be executed while the value of the real variable x was between 0.0 and 10.0. You may know that, in mathematics, this condition can be expressed as "$0.0 \leq x \leq 10.0$". Translating this into Pascal, one might write

```
while 0.0 <= x <= 10.0 do ...  {illegal in Pascal}
```

As noted, this gives a syntax error. Why? The two <= operators in this expression are of equal precedence, so they are evaluated left to right. The result of the first comparison (0.0 <= x) is a Boolean value, TRUE or FALSE. Thus, when Pascal tries to carry out the second <= operation, the value on the left side of the operator is a Boolean, and the value on the right is a real (10.0). This, of course, is not the original intent of the expression. Worse, as far as Pascal is concerned, is that the expression winds up trying to compare a Boolean with a real. Pascal will refuse to do it, displaying its usual "type incompatibility" bug box.

You might try again, writing the comparison more or less as you might say it: "while x is greater than or equal to 0.0 and less than or equal to 10.0...." This might translate into Pascal as follows:

```
while x >= 0.0 and <= 10.0 do ...  {also illegal in Pascal}
```

Unfortunately this is also wrong; all relational operators are binary operators and they need values to compare on both sides. The <= operator has 10.0 on its right side but no value on its left. The same argument applies to the **and** operator, which needs to work with two Boolean values on either side.

Whenever you write two binary operators side by side, it is a signal that you are doing something Pascal will not allow.

A reasonable next attempt at the solution might be the following:

```
while x >= 0.0 and x <= 10.0 do ...  {also illegal in Pascal }
```

This avoids all the problems discussed before; it is still not quite right, however. A quick glance at the precedence chart in Figure 2-10 shows why. There are three operators in the expression; of the three, the **and** has highest precedence. So Pascal will try to carry out the **and** operation first. The problem is that the values on either immediate side of the **and** are reals, but **and** only operates on Boolean values. The result is, once again, the "type incompatibility" bug box.

The correct way to express the condition, then, is to put both comparisons in parentheses to ensure they are done before the **and** operator tries to operate on the results:

```
while (x >= 0.0) and (x <= 10.0) do ...
```

Here the comparisons are performed first, each giving a Boolean result; these Boolean values are then combined with the **and** operator to arrive at the correct Boolean result.

If you feel you need more experience with expression evaluation, try typing in the following program, which simply prints out the values of various expressions. Try to predict what the output results will be before you run the program. If any of your predictions were wrong, try to figure out where you and Pascal disagreed in your evaluations.

```
program expression_lab;
{ experiments with expressions }

var
 x, y, z : real;
 i, j, k : integer;

begin { expression_lab }
 x := 3.6;
 y := 2.4;
 z := 0.6;
 i := 32;
```

```
    j := 7;
    k := 2;
    writeln(x + y / z : 15);
    writeln((x + y) / z : 15);
    writeln(-x * y / z : 15);
    writeln(x / y / z : 15);
    writeln;
    writeln(i - j - k : 1);
    writeln(i - (j - k) : 1);
    writeln(i mod j div k : 1);
    writeln(i - k mod j : 1);
    writeln;
    writeln((i >= j) and (j >= k));
    writeln(not (i >= x) or not (j >= y));
    writeln((i <> i) or (j = j));
    writeln(not (not (not (not (not (k > i))))))
  end.
```

As usual, you should feel free to compose your own complex expressions and verify that they are evaluated under the rules discussed here.

MACINTOSH PASCAL: EDITING AND DISK USE

3

> Things that are done well tend to be transparent and are not noticed, while things that are done badly intrude and are noticed.
>
> —*Paul Heckel*
> *The Elements of Friendly Software Design*
> (Warner Books, 1984)

In Chapter 1, we focused on how to use Macintosh Pascal, discussing how to type in and modify simple programs. You gained more practice using Macintosh Pascal in Chapter 2, but the emphasis in that chapter was on the Pascal language itself, rather than the nuts-and-bolts details of how Pascal works on the Macintosh. In this chapter we will return to Macintosh-only material; the main concern will be program editing and how it can be made even easier and faster. We will also discuss some housekeeping techniques for disks; these will become useful once you start developing more and larger programs.

SELECTION

A ubiquitous concept in all good software for the Mac is that of *selection*. If you have used MacPaint, for example, you know that you may select any area in your drawing with either the selection rectangle or lasso. In MacWrite, you may select a part of your document using the mouse. We discussed briefly

in Chapter 1 how to select icons on the Macintosh desktop by clicking them. In general, the term "selection" refers to telling the Macintosh what you want to work on. The information selected may then be acted on as a unit.

Macintosh Pascal uses selection mainly to make program editing easier. Until now, your editing has been restricted to inserting or deleting one character at a time. You may have wished that there were some special commands that allowed you to delete a ten-character identifier, for example, other than the method you now use (moving the insertion point to the end of the identifier and pressing BACKSPACE ten times). Once you learn the selection methods presented here, you will have a number of such special commands to choose from.

If you have some MacWrite experience, you'll be glad to know that the selection methods used in Macintosh Pascal are very similar. There are a few minor differences, though, so even MacWrite experts should at least skim this section.

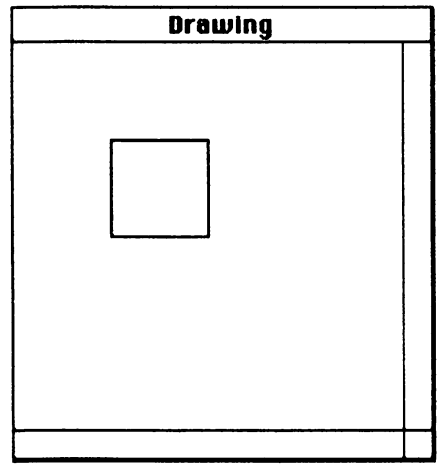
To see how selection works, type in the following program:

```
program quad;
{ Draw a quadrilateral }

begin { quad }
 moveto(50, 100);
 lineto(100, 100);
 lineto(100, 50);
 lineto(50, 50);
 lineto(50, 100)
end.
```

This simple program uses two Macintosh Pascal commands that we haven't discussed yet: *moveto* and *lineto*. As you might suspect, these commands allow you to use the Macintosh's graphics capabilities. The output from this program will appear not in the Text window but in the Drawing window. The Drawing window appears automatically when Macintosh Pascal is started, but in the course of programming, it is easy to move or change the size of the Program or Text window so that the Drawing window is partially or entirely covered. If this happens, resize the windows before you run the program so that you can see the entire Drawing window. If you have closed the Drawing window by clicking its close box, you may

reopen it by choosing the Drawing option from the Windows menu.

In any case, once you can see the Drawing window, run the program you typed in. You should see a square on your screen:



The five commands in the program just entered draw the four sides of a square. (Consider this to be a sneak preview of the Macintosh Pascal QuickDraw routines, which will be explained in Chapters 8 and 12.)

You might wonder about the meaning of the numbers placed in parentheses after the command names. To try to discover what they're for, let's modify the program. A typical experiment might be to change the second number in the first lineto command from 100 to 5. Instead of deleting or inserting one character at a time, however, let's select the number to be changed. To accomplish this, do the following: (1) position the I-beam pointer in front of the second 100 in the first lineto command, (2) press and hold the mouse button, (3) drag the pointer across the number to highlight it, and (4) release the mouse button. The line now looks like this:

```
lineto(100, 100);
```

Congratulations: you have successfully *selected* a piece of program text. If you don't get this result the first time, click the mouse button once, which will undo the selection you

made, and try again. Accurate selection may take a little practice, but you'll find it a worthwhile skill to master.

We'll now assume you have successfully selected the number 100. To change the number 100 to 5, just press the 5 key. The highlighted 100 vanishes and is replaced by a 5. Run the program again to see how this changes the drawn figure.

You have just seen the first example of how selection can make editing easier. In general, to replace something in your program with something else, all you need do is select the text you want to change and then type in the replacement text. The first keystroke of the replacement text deletes the original text you selected. If you must replace more than a few characters of your program, this will be considerably faster than deleting the original text one character at a time using the BACKSPACE key.

Some thought should tell you that selection is a potentially risky technique. If you accidentally select a large amount of text and then press a key, the entire selection can be erased from the screen. Unless you have saved the program on disk, this text is gone for good; the only way to get it back is to retype it. Macintosh Pascal, unlike MacWrite and MacPaint, has no undo command to allow recovery of accidentally deleted material. This doesn't mean you shouldn't use selection; it simply means you must be careful when using it.

You may select any part of your program using the same technique: just move the pointer to the beginning of the text you want to select, press and hold the mouse button, drag to the end of the desired selection, and then release the button. The selection isn't completed until you finally release the button, and any selection you make can be immediately undone by clicking once anywhere inside the Program window.

You might want to practice selecting any of your program text using this method. In addition to selecting within a line, you can select text starting in one line and ending in another. Also attempt to select the entire program. (An easy way to select multiple lines in a program is to drag down the left margin of the window.) After you make each selection, undo it by clicking with the pointer anywhere in the Program window.

Macintosh Pascal provides two shortcuts for faster and more accurate selection of words and lines. To select a single word or number in your program, just double-click the word or number. To select an entire program line, triple-click the line.

Another selection shortcut is commonly known as *shift-click* selection. To see how it works, first move the insertion point to one end of a desired selection. Then move the pointer to the other end. Hold down the SHIFT key and click the mouse button; this will select all program text between the insertion point and the pointer. Shift-click also works to extend a previous selection. Select a small amount of the program, move the pointer to another point, and shift-click. The selection will extend from the original selection to the location of the pointer.

An additional shortcut is available if you want to select your entire program: choose Select All from the Edit menu. You'll want to become accustomed to these shortcuts, so practice them as well as the more general method of dragging the pointer.

Instead of replacing text, you may want to simply delete text from your program. You can use selection to do that as well: just select the text you want to delete and then either press the BACKSPACE key or select the Clear option from the Edit menu. (Pressing the BACKSPACE key is probably faster.) Once again, you should be careful not to accidentally delete a large chunk of your program, especially if the program has not been saved on disk; Macintosh Pascal provides you with no recovery from a mistake like this.

For experience in selecting, make the following changes to your program, running it after each change. Try to accomplish each change with the minimum number of keystrokes:

- Undo the change you just made to the program: change the line containing "lineto(100, 5)" back to "lineto(100, 100)".
- Change the line "lineto(50, 50)" to "lineto(75, 50)".
- Change the line "lineto(100, 50)" to "lineto(125, 50)".
- Change the line "lineto(50, 100)" to "lineto(25, 100)".
- Change the line "moveto(50, 100)" to "moveto(25, 100)".

CUTTING, PASTING, AND COPYING

Selection would be useful even if all we could do with it was what we've discussed so far. But selection is also the basis for additional powerful editing operations: cutting, copying, and

pasting. We'll explore these operations in this section. Cutting, copying, and pasting operations are, like selection, common techniques used in nearly all Macintosh software.

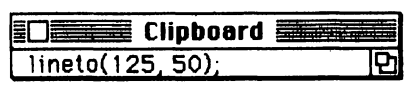
Once you select a piece of your program, you may choose to cut it out of your program. Like deletion, this removes the text from your program. Unlike deletion, however, it saves the text for possible reuse. The place where a cut is saved is called the *Clipboard.* The Clipboard is used by most Macintosh software; it is used to hold just one piece of selected program (or drawing or document).

At this point in the discussion, your program should look like this:

```
program quad;
{ Draw a quadrilateral }

begin { quad }
 moveto(25, 100);
 lineto(100, 100);
 lineto(125, 50);
 lineto(75, 50);
 lineto(25, 100)
end.
```

Select the line "lineto(125, 50);". (Remember, the easiest way to do this is to triple-click the line.) Cut the line from the program by choosing the Cut option from the Edit menu. The line should vanish from your program. (Run the program to verify if you like.)

To confirm that the line was not simply deleted, choose the Clipboard option from the Windows menu. A new window labeled Clipboard will appear on your screen and become the active window; the window will show the line you cut:



We have cut a single line here, but as you work this example, remember that we could just as well have cut any contin-

uous region from the program, ranging from a single character all the way to the entire program itself. The Clipboard imposes no arbitrary limits on the size of the text you store there. If the text is small enough, it will be saved in memory. Otherwise, it will be stored in a *Clipboard file* on disk. Cutting only fails if there is no room for the selected text either in memory or on your disk.

In general, you only need to look at the Clipboard if you can't remember what it was you stored there. As with most windows, you may move the Clipboard around the screen and change its size (although it can't be scrolled). For now, close the Clipboard window by clicking its close box.

One cautionary note about the Clipboard: it only stores one thing at a time. If you perform two successive cut operations, the information on the Clipboard from the first cut is obliterated by the second. It is a good general rule to retrieve from the Clipboard immediately after a cut to avoid accidentally losing important program text.

Retrieving text from the Clipboard is called *pasting.* Whenever you paste, everything on the Clipboard is copied into your program at the current insertion point. To see this happen, move the insertion point to just before the line "lineto(100, 100);", then choose the Paste option from the Edit menu. Instantly, the line you previously cut appears in a new place. (Again, if you wish, run the program to see how this change affects the drawing.)

Pasting does not change the contents of the Clipboard. To verify this, move the insertion point so that it precedes the line "lineto(75, 50)" and choose the Paste option again. Once more, the line on the Clipboard is copied into the program at the insertion point, just as if you had typed it yourself.

When you edit larger programs, you will most often use cutting and pasting for moving part of your program from one place to another. As you have seen, this is a simple process: cut the text you want to move and paste it into the new position.

Now you know about cutting and pasting; what about *copying?* Essentially, copying works just like cutting, except that the selected text is not removed from the program; only a copy of the text is moved onto the Clipboard.

To see for yourself how copying works, select the line "moveto(25, 100);" in your current program. To copy this line onto the Clipboard, select the Copy option from the Edit

menu. This operation will not affect your program at all; the only change will be to the Clipboard contents. If you want, choose the Clipboard option from the Windows menu to verify that the line has been copied there. After that, either close the Clipboard window or activate the Program window by clicking it.

Now move the insertion point to before the line "lineto(100, 100);" and choose the Paste option from the Edit menu. Once again, the text on the Clipboard is pasted into the program at the insertion point. Just as you can use a cut-and-paste operation to move a part of your program from one place to another, you can use a copy-and-paste operation to make a copy of any program segment at another position.

Here are two practice exercises; as usual, run your program after each change you make:

- Make a copy of the line "lineto(100, 100);" just after the line "lineto(75, 50);".
- Make a copy of the line "lineto(75, 50);" just before the line "lineto(25, 100);".

In addition to copying text from one part of a program to another, the copy-and-paste operation allows you to transfer program text from one of your programs to another. The following steps will do the job:

- Copy the text from the source program into the Clipboard.
- Close the source program window by choosing Close from the File menu.
- Open the destination program window by choosing Open from the File window and choosing the destination program's name.
- Paste the Clipboard contents into the desired place in the destination program text.

In addition to program-to-program transfers, you may also copy and paste to and from documents used by other applications. For example, you may occasionally find it useful to prepare your programs using a word processing program like MacWrite; to do this, simply select and copy the program text to the Clipboard from the MacWrite document, quit

MacWrite, run Macintosh Pascal, and paste the Clipboard into the Program window. You can use the reverse process to copy Pascal program segments into MacWrite documents. (Be aware that some information contained in a document is often lost in a copy-and-paste operation between applications. For example, in a Pascal-to-MacWrite transfer, Pascal reserved words lose their usual boldface.)

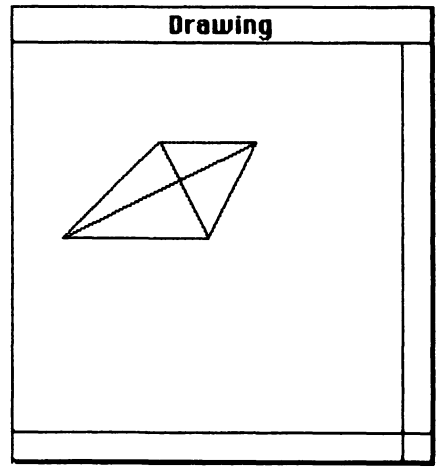
FINDING AND REPLACING

Two more editing operations that will enable you to modify your programs easily and reliably are *finding* and *replacing*. Finding refers to the action of searching the program for a specific sequence of characters. Replacing refers to finding text and then replacing that text with another sequence of characters. In small programs it is easy to find and replace text yourself using the techniques we've already discussed. In larger programs, however, it is much easier to turn the task over to the computer. The computer can also perform the task without mistake, a feat that's not so easy for humans.

Our program should now look like this:

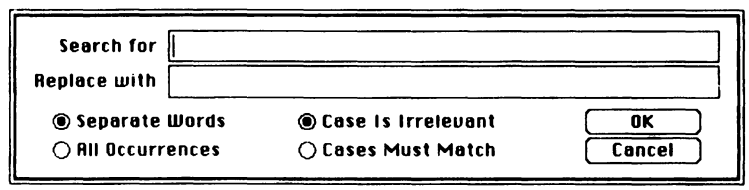
```
program quad;
{ Draw a quadrilateral }

begin { quad }
 moveto(25, 100);
 lineto(125, 50);
 moveto(25, 100);
 lineto(100, 100);
 lineto(125, 50);
 lineto(75, 50);
 lineto(100, 100);
 lineto(75, 50);
 lineto(25, 100)
end.
```

When run, the program produces the following drawing.



You might decide that "quad" is no longer an adequate name for this program; the output looks more like a kite. To change the name of the program you could use editing operations we have already discussed, but this time let's use the find and replace operations.

Finding and replacing are two-step operations; first, you must tell the Macintosh what text to find and, optionally, what to replace it with. Once you have done that, you can actually perform the find or replace. To get started, choose the What to find... option from the Search menu. (Note the What to find... option is the only undimmed option in the menu.) This dialog box will appear after some disk whirring:



The blinking insertion point is in the box labeled "Search for"; as usual, any characters you type will appear at the insertion point. Type the word "quad", which is the string we want to search for, and then click the OK button, which will make the dialog box vanish.

Next choose the now undimmed Find option from the Search menu. There are two occurrences of the word "quad" in the program, and it will find one of them; you will notice

that it will be highlighted. Which one will be found? It depends; the search proceeds forward from the current position of the insertion point. The search will *wrap around*, if necessary, from the end of the program to the beginning.

Choose the Find option two or three times in a row to verify that the next occurrence of "quad" is always found. Text that is found is always automatically selected.

You may also notice that the occurrence of "quad" in the word "quadrilateral" is not found. This is because, unless you specify otherwise, Macintosh Pascal will only search for "words" in your program—that is, text set apart by spaces or punctuation of some kind. Choose the What to find... option from the Search menu once more, and this time click the All Occurrences button and then the OK button. Finally, choose Find (more than once, if necessary) to verify that, this time, the partial word is also found.

Normally the case of the text in your program will not matter to a search; if you search for the word "apple," for example, you will find "Apple," "APPLE," and even "aPpLe." If you only want to find exact matches, including case, click the Cases Must Match button in the What to find... dialog box before you click OK.

A few minor points: you may search for text appearing on more than one line; simply type a space where the linebreak occurs. You are not limited to searching for text that will fit in the "Search for" box, although you will, unfortunately, only be able to see the text that fits. Finally, all the familiar insertion, deletion, selection, cutting, copying, and pasting operations work when you type text into the dialog box.

Let's now replace our found text: choose the What to Find... option to display the dialog box once more. Click in the box labeled "Replace with" to move the insertion point down there; then type the word "kite." Click OK, as usual. (Clicking the Cancel button will ignore the changes you made to the information previously stored in the dialog box.)

Now choose Find once more to highlight an occurrence of "quad"; then choose the Replace option from the Search menu. The selected occurrence of "quad" will be replaced by "kite," as requested.

Finally, choose the remaining option in the Search menu: Everywhere. As you might guess, this performs the replacement specified in the dialog box everywhere in your program. Since this is a rather drastic operation, Macintosh Pascal will display a dialog box asking you to confirm whether

want to go ahead with it. (Click the Yes button in this case.) Your resulting program should appear as follows:

```
program kite;
{ Draw a kiterilateral }

begin { kite }
 moveto(25, 100);
 lineto(125, 50);
 moveto(25, 100);
 lineto(100, 100);
 lineto(125, 50);
 lineto(75, 50);
 lineto(100, 100);
 lineto(75, 50);
 lineto(25, 100)
end.
```

Note that, in addition to "quad" being changed into "kite," the word "quadrilateral" has been transformed into "kiterilateral." This interesting behavior is a result of our clicking the All Occurrences button in the dialog box; if the Separate Words button had been clicked instead, Macintosh Pascal would have only replaced "quad" where it appeared as a whole word.

In normal program editing, you will probably specify both the Search for and Replace with strings in a single visit to the dialog box; you will also want to make the appropriate button choices then as well.

Like all operations that specify changes to your program, replacing can be risky, especially Everywhere replacements. The main danger is that you will change something you didn't intend to. To minimize accidents, always try to specify a Search for string that only matches the parts of the program you really want to change. If possible, use the Separate Words button to change only whole words. If you have doubts about whether you will accidentally change something, run through the program with the Find option and peruse each selection before you use Everywhere.

Once again, here are some practice exercises:

- Using the Separate Words button, change all occurrences of "25" in the program to "50".
- Change all occurrences of "125" to "100".
- Change all occurrences of "75" to "50".

What happens when you run the program now? (You may want to change the name again.)

SHORTCUTS

So far we have discussed numerous techniques that make the job of typing in and modifying Macintosh Pascal programs easier. In this brief section, you'll see a number of other techniques—shortcuts—that make the editing tasks previously discussed even easier and faster to carry out.

In most Macintosh software there are often two ways to accomplish common operations: the first method will be easy to remember and the second will be easy to do (and perhaps not as easy to remember, at least at first). The second method is the *shortcut:* in general, a shortcut will be a slightly faster way to do something. Each shortcut you use will typically cut only a second or two from the time it takes to do the operation. But since the shortcuts are common operations, you'll find that their frequent use makes your work go noticeably faster.

We have already discussed a number of shortcuts in passing: they include double-clicking and triple-clicking to select a word or a line. Shift-click selection might be considered a shortcut, and the cut, copy, paste, find, and replace operations can all be considered shortcuts since their results could be accomplished using more time-consuming methods.

Shortcuts often involve pressing the COMMAND key at the same time you press one or two other keys. The COMMAND key is the one to the immediate left of the spacebar on your Macintosh keyboard, marked with a cloverleaf. When used in combination with other keys, the COMMAND key works like the SHIFT key: the rules are that you must press the COMMAND key before you press the other keys and continue holding it while you press the other keys. (Although this process may sound involved, it's one you always do whenever you use a SHIFT key for uppercase. A little practice will make the COMMAND key equally easy to use.)

With the COMMAND key, you can activate many menu options directly from the keyboard instead of pulling down the menu with the mouse and dragging down to the desired option. You have probably noticed the cloverleaf and letter symbols following some menu options; these show what the

Table 3-1.

Edit Menu Shortcuts

Menu Option	Shortcut	Action
Cut	COMMAND-X	Move selection to Clipboard
Copy	COMMAND-C	Copy selection to Clipboard
Paste	COMMAND-V	Copy text from Clipboard to program at insertion point
Clear	BACKSPACE	Delete selection
Select All	COMMAND-A	Select entire program

equivalent keyboard commands are. For example, to run your program, instead of choosing the Go option from the Run menu, you may hold the COMMAND key and type the G key. (To use the shorthand way of referring to this key combination, type a COMMAND-G.) Run your program now by pressing COMMAND-G.

COMMAND key shortcuts used in Macintosh Pascal are shown in Tables 3-1, 3-2, and 3-3. Table 3-1 shows the shortcuts used to access Edit menu options and Table 3-2 shows the Search menu shortcuts. Table 3-3 lists the shortcuts for common Run menu options; these include two options we will discuss in Chapter 5.

Here are two previously unmentioned shortcuts that apply generally to many Macintosh applications:

Table 3-2.

Search Menu Shortcuts

Menu Option	Shortcut	Action
Find	COMMAND-F	Find next occurrence of "Search for" text
Replace	COMMAND-R	Replace next occurrence of "Search for" text with "Replace with" text
Everywhere	COMMAND-E	Replace all occurrences of "Search for" text with "Replace with" text
What to find...	COMMAND-W	Set "Search for" and "Replace with" texts

Table 3-3.

Run Menu Shortcuts

Menu Option	Shortcut	Action
Check	COMMAND-K	Check program for syntax errors
Go	COMMAND-G	Run program
Step	COMMAND-S	Execute one program line

- Pressing the RETURN or ENTER keys in a dialog box is usually equivalent to clicking either the OK button or the outlined (default) button.
- Pressing the TAB key in a dialog box will skip from the insertion point to the next text entry position.

Don't force yourself to memorize these shortcuts. You will find yourself naturally using them more and more as you gain experience with Macintosh Pascal. And that experience will come soon enough.

MACINTOSH PASCAL HOUSEKEEPING HINTS

The Macintosh Pascal software supplied by Apple Computer contains two disks, one a copy of the other. The disks contain a number of interesting demonstrations, some utility programs, and (most importantly, perhaps) some documentation that didn't make it into the printed manuals provided with the system. If you haven't yet done so, run the demonstration programs, find out how the utility programs are used, and at least look through the documentation. A good place to start is the program called Open Me; its use is self-explanatory.

You may, for archival purposes, easily make backup copies of every file on these disks except for the Macintosh Pascal application itself. The Macintosh Pascal file is copy-protected; it can't be successfully duplicated on another disk using normal Macintosh copying operations. Some programs that help you make archives will copy the Macintosh Pascal file, even though it is copy-protected. For example, Copy II Mac from Central Point Software will copy the entire disk correctly.

Frankly, this copy-protection makes Macintosh Pascal

harder to use than it would be otherwise. The primary purpose of this section is to offer suggestions on how to best deal with this situation. In general, the methods suggested here will allow you to use Macintosh Pascal with minimal access to the Macintosh Pascal disk itself, decreasing wear and tear on the valuable disk. We will also discuss different strategies for saving and retrieving your own programs to and from disk.

If, despite your best efforts, either one of your Macintosh Pascal disks becomes unusable, follow the instructions included with the software to obtain another copy, or see your local Apple service center.

Our first topic is the preparation of working disks from the supplied Macintosh Pascal disks. The following discussion assumes you have some familiarity with the Macintosh Finder. If you are a novice, perhaps you should review the Macintosh documentation or get an experienced friend to help you in this process. A little carelessness during the following steps can, unfortunately, make your valuable disks worthless. (Don't be scared, just be careful.)

The first thing you should do is put one of the copies in a safe place. It is not too extreme to put it in a different building from the one you are working in, since disks are easy to lose to theft, smoke, or water.

The next thing you should do is make at least two copies of the remaining Macintosh Pascal disk (which we will call the Master Disk). This will back up all files from the Master Disk except the Macintosh Pascal application itself. You cannot use the normal icon-dragging method to make the copies since the Macintosh Pascal file is protected. The Disk Copy application won't work either. To make the copies, which will be called the Auxiliary Files disks, follow these steps:

- While in the Finder, close any open windows and eject any disks. Insert both the Master Disk and a blank disk. (On a single-drive system, insert one disk, eject it, and insert the other.) If the new disk has never been initialized, you'll be prompted to do so.
- Open each disk's icon by double-clicking the icon. Select all of the files and folders in the Master Disk's window except the Macintosh Pascal application (the icon of hands on the keyboard) and the Empty Folder icon.
- Drag the selected files and folders into the window of the Auxiliary Files disk. On a single-drive system, you will be prompted to swap the original and new disks a

number of times. When the copying is finished, you have created an Auxiliary Files disk.

- Repeat these steps for a second Auxiliary Files disk.

Delete everything from your Master Disk except for Macintosh Pascal and the System Folder by dragging all other icons to the Trash. (This is safe since you have at least three other copies of the files you are deleting here if you performed the previous steps. If you haven't been following the discussion, *don't* start here. Go back to the beginning of this section.) If you accidentally drag either the System Folder or Macintosh Pascal to the Trash, open the Trash icon and drag the Folder or Pascal back to the disk window immediately.

To repeat, your Master Disk will now contain only the System Folder and Macintosh Pascal. This is all you require for normal use of Macintosh Pascal. To minimize reading from and writing to this valuable disk, these should be the only items stored there. (Unfortunately, you can't take the even safer course of write-protecting the disk, as Macintosh Pascal needs to be able to store data on the disk while printing and during some cut-and-copy operations.)

This suggested organization poses no problems if you use a Macintosh with two disk drives. When you use Pascal, keep the Master Disk in the internal drive and a program-storage disk in the external drive. When you save a program you have written, be sure you save it to the program disk, not the Master Disk. All your programs may be loaded from the external drive as well.

If you are a single-drive user, the process of saving a program will be slightly more complex. You will still need to keep the Master Disk in the drive while you are running Pascal. To store a program you have written on a single-drive system, follow these steps:

- Choose the Save as... option from the File menu.
- When the Save as... dialog box appears, click the Eject button to temporarily eject the Pascal disk.
- Insert the disk on which you want to save the program.
- Type a name for the program in the dialog box and click the Save button.
- Once the program has been safely saved, the disk will be automatically ejected and you'll be prompted to reinsert the Pascal disk.

Retrieving a program from another disk on a single-drive system follows a similar process:

- Choose the Close option from the File menu to clear memory for the program. (Save the current program in memory first, if you want.)
- Choose the Open... option from the File menu.
- When the dialog box appears, click the Eject button to temporarily eject the Pascal disk.
- Insert the disk containing the program you want to retrieve.
- Open the desired program by either clicking its name and then the Open button or, as a shortcut, double-clicking its name.
- Once the program has been read from the disk, you will be prompted to reinsert the Master Disk.

As your collection of Pascal programs grows, the way in which you store the programs on your disks becomes more important. Unfortunately, the importance of good disk habits is often only realized when catastrophe strikes.

Although the precise organization you use will depend on your own special situation, you will probably find it best to keep closely related programs grouped in folders and to keep related folders grouped (when possible) on the same disk.

Every disk that contains even slightly important programs or documents should be backed up on a regular basis; often you'll want to back up a disk whenever you add or change any information on it. You will also find it useful to label your disks clearly and store them in a logical manner so that you can rapidly find any program or document. Most importantly, your labels should clearly distinguish original disks and backups. (What happens if you copy a backup disk to an original disk?)

DECISION MAKING

4

The general rule is: after you make a decision, *do something.*

Brian Kernighan and P.J. Plauger,
The Elements of Programming Style
(McGraw-Hill, 1978)

In Chapter 2, we discussed the Pascal **while**, **repeat**, and **for** looping statements. In this chapter, we will cover the *decision* statements available to your Pascal programs. These statements don't involve repetition; instead, they allow your program to take action based on conditions that develop while the program executes.

THE if...then STATEMENT

The simplest decision your program can make is accomplished with the **if...then** statement. The syntax sketch for this statement is shown in Figure 4-1. When your program encounters an **if...then**, the Boolean expression between the **if** and the **then** is evaluated; if the expression has the value TRUE, the statement following the **then** is executed. If, on the other hand, the expression evaluates to FALSE, the statement is skipped.

Figure 4-1.

if...then statement syntax

A simple example of the **if...then** statement is the following:

```
program if_lab;
{ Experiments with if statements }

 var
  x, y : real;

begin { if_lab }
 x := 1.2;
 y := 3.4;
 write('w');
 if x < y then
  write('ind');
 write('ow')
end.
```

Run Macintosh Pascal and type in this program. Before you run the program, though, make an educated guess at the output.

The Boolean expression evaluated by the **if...then** is:

```
x < y
```

If this expression evaluates to TRUE, the statement

```
write('ind')
```

will be performed. If the expression evaluates to FALSE, the statement will not be performed. The statements preceding and following the **if...then** are executed in any case, of course.

In the example, the Boolean expression evaluates to TRUE because the value of the variable x, 1.2, is less than the value of the variable y, 3.4. So all three write commands are executed, and the output is the word "window."

Now change the line

```
x := 1.2
```

to

```
x := 5.6
```

Again try to figure the result before you run the program. Now the Boolean expression will evaluate to FALSE (because 5.6 is not less than 3.4), so the statement bracketed by the **if...then** will not be performed. Only the two write commands above and below the **if...then** will be done, making the output this time simply "wow."

Once more the general rule of Pascal statements applies: you may place any Pascal statement inside the **if...then**, including compound statements, **while** statements, and so on. You may also legally place the **if...then** itself anywhere any other statement could go. Save the if_lab program on disk; we'll return to it later.

In another example, we'll modify the square root calculator developed in Chapter 2:

```
program Newton;
{ Calculate square roots by Newton's method }

 var
  x, old_guess, new_guess : real;

begin { Newton }
 write('Enter a number:');
 readln(x);
 new_guess := x / 2;
 repeat
  old_guess := new_guess;
  new_guess := (old_guess + x / old_guess) / 2
 until new_guess = old_guess;
 writeln('The square root of ', x : 16, ' is ', new_guess : 16)
end.
```

Either retrieve the program Newton from disk (if you saved it) or type it in.

You may remember that Newton was an unreliable program because it crashed when fed a zero value for the variable x and went into an infinite loop when given a negative number for x. To make the program more reliable, we can use the **if...then** to protect the calculation from unexpected values. Modify the program as follows:

```
program Newton;
{ Calculate square roots by Newton's method }

var
 x, old_guess, new_guess : real;

begin { Newton }
 write('Enter a number:');
 readln(x);
 if x > 0.0 then
  begin
   new_guess := x / 2;
   repeat
    old_guess := new_guess;
    new_guess := (old_guess + x / old_guess) / 2
   until new_guess = old_guess;
   writeln('The square root of ', x : 16, ' is ', new_guess : 16)
  end
end.
```

Don't forget to insert the new **end** down near the bottom of the program; this matches the **begin** after the **if**. Now run the program again. You'll find it still works when you give it positive values and that it doesn't behave badly now when it reads zero or negative numbers (actually, it doesn't do anything at all). In this case, the statement controlled by the **if...then**

```
begin
 new_guess := x / 2;
 repeat
  old_guess := new_guess;
  new_guess := (old_guess + x / old_guess) / 2
 until new_guess = old_guess;
 writeln('The square root of ', x : 16, ' is ', new_guess : 16)
end
```

is a compound statement. This compound statement contains a repeat loop, which itself contains two assignment statements. This is just another example of how Pascal statements can be nested within each other, a concept you will see over and over in nearly all significant Pascal programs.

THE if...then...else STATEMENT

In the previous section you saw it was possible to do something based on the value of a Boolean expression. It is often more desirable that, in addition to doing something when the Boolean expression is TRUE, your program can do something else when the Boolean expression is FALSE.

Assuming you have been following this discussion on your computer, save your new version of the Newton program on disk and retrieve the if_lab program you saved in the previous section. Modify if_lab to look like this:

```
program if_lab;
{ Experiments with if statements }

 var
  x, y : real;

begin { if_lab }
 x := 5.6;
 y := 3.4;
 write('w');
 if x < y then
  write('ind')
 else
  write('all');
 write('ow')
end.
```

Notice that the write statement after the **if...then** is no longer followed by a semicolon; make sure your program looks the same. (We will discuss the missing semicolon shortly.)

What will happen when you run the program? (As usual, try to guess first.) The Boolean expression will evaluate to FALSE, so the statement following the **then** won't be executed. Instead of doing nothing in this case (as happened with

the simple **if...then**), the statement following **else** is performed. The result is the word "wallow." Now change the line

```
x := 5.6
```

back to

```
x := 1.2
```

What will happen now?

To summarize, the **if...then...else** gives your program two possible actions to take based on the result of evaluating a Boolean expression; the syntax sketch is shown in Figure 4-2. If the expression gives a TRUE result, the statement following **then** is executed; if FALSE, the statement following the **else** is executed. (And although you may be tired of hearing this, the statements may be any Pascal statements, including compound statements.)

Another important point to remember about the **if...then ...else** is that exactly one of the statements will always be performed, no matter what. Since the Boolean expression must be either TRUE or FALSE, there is no way that both statements (or neither statement) will be executed.

Why was it so important that you remove the semicolon from the statement preceding **else**? To see why, try putting the semicolon back in, as follows:

```
program if_lab;
{ Experiments with if statements }

 var
  x, y : real;

begin { if_lab }
 x := 1.2;
 y := 3.4;
 write('w');
 if x < y then
  write('ind'); { an error about to happen }
 else
  write('all');
 write('ow')
end.
```

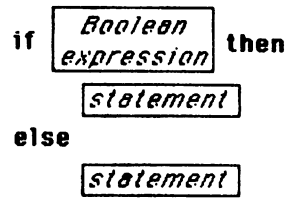
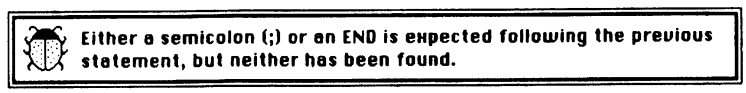


Figure 4-2.

if...then...else statement syntax

You need not type in the comment, of course. Note that the word **else** turns into an outline font once you type the semicolon, which is Pascal's way of warning you that syntax errors are imminent. When you try to run the program, the result is this bug box:



You may note that the bug box isn't especially illuminating as to the cause of the syntax error. (We inserted an extra semicolon, but the error message seems to think there are too few.) The problem lies in our carelessness with Pascal's simple but absolute semicolon rule. Although we alluded to this rule in Chapter 1, it is worth emphasizing here:

Semicolons separate Pascal statements

Since semicolons are statement separators, Pascal always expects to see a complete statement both preceding and following a semicolon. An **else** is not a legal way to begin a Pascal statement since it is not a statement itself, only a reserved word that is *part* of a possible statement. So Pascal gives up and reports a syntax error at this point in the program.

It may seem to you that semicolons are more difficult to understand than any other part of the Pascal language. You have seen that semicolons are mandatory at certain points of

your program (between any two statements, for example) and, as in the last example, are prohibited at other places (before an **else**, for example). What can confuse things slightly is that at certain points of your program semicolons can be *optional.*

To see an example, insert a semicolon in the if_lab program just before the final **end**. (Don't forget to remove the bad semicolon first.)

```
program if_lab;
{ Experiments with if statements }

var
 x, y : real;

begin { if_lab }
 x := 1.2;
 y := 3.4;
 write('w');
 if x < y then
  write('ind')
 else
  write('all');
 write('ow');
end.
```

Run the program. As you can see, Pascal has no problem with this new semicolon at all. Everything works as before.

But if you give this a little thought, you may object that the new semicolon cannot be a statement separator because the word **end** is not a Pascal statement, just as **else** was not a Pascal statement. Doesn't this behavior break the semicolon rule?

The answer is no; the rule is still valid, but the reasoning is a little subtle. When you add the semicolon, Pascal considers that there has to be an *empty statement* between the semicolon and the word **end**. That way the semicolon really is separating two statements: the write statement and an empty statement.

The syntax sketch for the empty statement is shown in Figure 4-3. It's drawn as an empty box. The box, of course, is not considered part of the statement; the empty statement is *inside* the box: it is nothing at all.

Lest you think this discussion of empty statements is hope-

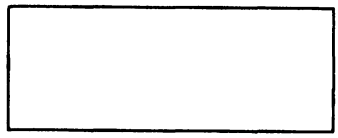


Figure 4-3.

Empty statement syntax

lessly pedantic, insert a semicolon after the **else** in the if_lab program:

```
program if_lab;
{ Experiments with if statements }

 var
  x, y : real;

begin { if_lab }
 x := 1.2;
 y := 3.4;
 write('w');
 if x < y then
  write('ind')
 else
  ;
 write('all');
 write('ow');
end.
```

Do you think this semicolon is illegal? Well, it may look illegal, but it is perfectly fine with Pascal. In the modified program, the statement controlled by the **else** is the empty statement between the **else** and the new semicolon. Fortunately, Macintosh Pascal reformats the program automatically, showing that the statement

```
write('all')
```

is no longer inside the **if...then...else** statement and so will always be executed no matter what the value of the Boolean expression. Run the program to confirm that's what happens.

(What will happen if you modify the program to make the Boolean expression evaluate to FALSE?)

So far, we haven't given you any good purpose for empty statements. Empty statements are occasionally useful when you want a statement to "do nothing." As an example, we will insert two empty **for** loops before and after the **if...then ...else**. (Note that the two **for** loops in this version of the program will be identical. Remember that you can use the copy-and-paste operation, so you only have to type one **for** loop.)

```
program if_lab;
{ Experiments with if statements }

 var
  x, y : real;
  i : integer;

begin { if_lab }
 x := 5.6;
 y := 3.4;
 write('w');
 for i := 1 to 1000 do
  ;
 if x < y then
  write('ind')
 else
  write('all');
 for i := 1 to 1000 do
  ;
 write('ow');
end.
```

Also note that the semicolon after the **else** has been removed. Finally, don't forget that the loop control variable i must be declared. Make sure these changes are in your program.

In this program the statement controlled by the **for** loops is the empty statement. Since the empty statement does nothing, this doesn't affect what the program does. But the **for** statements do take time to execute; in this case, the computer must count from 1 to 1000. Such do-nothing loops are often called *delay loops*, since they only slow the program down. (Run the program to find out how much slower. What happens when you change the 1000 to 10,000?)

In spite of its unobtrusive appearance, the empty statement is a normal Pascal statement, and it can be placed in your program anywhere you would place any other statement. In general, we would advise you to use empty statements only in those cases where they are specifically called for: those rare cases where *nothing* is exactly what you want your program to do, as in delay loops. Avoid extraneous semicolons, even when they are technically legal.

There is one final point you should be aware of. Modify the if_lab program once more:

```
program if_lab;
{ Experiments with if statements }

var
 x, y, z : real;

begin { if_lab }
 x := 5.6;
 y := 3.4;
 z := 7.8;
 write('w');
 if x < y then
  if y < z then
   write('ind')
  else
   write('all');
 write('ow');
end.
```

This is another example of statement nesting; this time, an **if** statement is nested within another **if**. With which **if** is the **else** associated? The simple rule is that an **else** always goes with the previous "un-elsed" **if**. Fortunately, Macintosh Pascal automatically indents your program in such a way to remind you of this rule.

Run the program to confirm that neither part of the "inner" **if...then...else** is performed; the output should be "wow." This is because the first **if...then** evaluates to FALSE.

If you want the **else** associated with the first **if** instead of the second, you must nest the second **if** inside a **begin...end** pair, as shown here.

```
program if_lab;
{ Experiments with if statements }

 var
  x, y, z : real;

begin { if_lab }
 x := 5.6;
 y := 3.4;
 z := 7.8;
 write('w');
 if x < y then
  begin
   if y < z then
    write('ind')
  end
 else
  write('all');
 write('ow');
end.
```

Make this modification and predict the output before you run the program. (Examine Macintosh Pascal's indentation of this program for a hint as to what will happen.) Also try changing the initial values of the variables x, y, and z in this program to print either of the two other possible words.

Once more let's return to the Newton program to improve it. (Again, save if_lab before you retrieve Newton.) You will remember that the previous version of Newton didn't do anything at all when it was handed a number it couldn't deal with; this can be disconcerting to someone who doesn't know what's happening. Add an **else** part to the **if** to print out an error message if the square root calculation can't be done:

```
program Newton;
{ Calculate square roots by Newton's method }

 var
  x, old_guess, new_guess : real;

begin { Newton }
 write('Enter a number:');
```

```
readln(x);
if x > 0.0 then
 begin
  new_guess := x / 2;
  repeat
   old_guess := new_guess;
   new_guess := (old_guess + x / old_guess) / 2
  until new_guess = old_guess;
  writeln('The square root of ', x : 16, ' is ', new_guess : 16)
 end
else
 writeln('Sorry, this program only works for positive numbers.')
end.
```

Make sure you can identify the two statements controlled by the **if...then...else** in this program; it's a little tricky because there are quite a few lines between the **if** and its matching **else**.

Pascal purists will object to the distinction made here between the **if...then** statement and the **if...then...else** statement. These are really not considered by Pascal to be two different kinds of statements. It is more accurate to refer to the **if...then...else** as a single statement type and to speak of the **else** part as optional.

THE else if STRUCTURE

To repeat a common theme, Pascal allows any statement to follow the **else** in an **if...then...else** statement—for example, another **if...then...else** statement. This second **if...then...else** statement may have a third **if...then...else** following its **else**, and so on. This process can be repeated as many times as desired, forming a series of **else ifs**. This structure is often called a *nested* **if...then...else if** *structure* or an **if...then...else** *chain*. We will call it an **else if** structure.

The **else if** structure is extremely common in Pascal programs, so common that the structure is worth considering in its own right. To see how it works, retrieve the if_lab program once more and modify as follows.

```
program if_lab;
{ Experiments with if statements }

var
 x, y, z : real;

begin { if_lab }
 x := 7.8;
 y := 3.4;
 z := 1.2;
 write('w');
 if x < y then
  write('all')
 else if x < z then
  write('ind')
 else if y > z then
  write('inn')
 else
  write('ill');
 write('ow')
end.
```

Note that the automatic formatting imposed by Macintosh Pascal on the if_lab program differs from what you might expect, based on your previous experience. Normally a statement following **else** starts on the line below **else** and is indented slightly. If the statement following the **else** is an **if...then**, however, the **if...then** follows the **else** on the same line. (As you see how the **else if** structure is used, you might want to speculate on Apple's reasons for this special treatment.)

Make a guess at the output from this program and run it to see if you were right. The program works as follows: in the first **if**, the Boolean expression

```
x < y
```

is evaluated. The result is FALSE, which means the statement following the **else** is to be executed. This statement is the second **if**, which means the Boolean expression

```
x < z
```

is evaluated. This is also FALSE, so the statement following

the second **else**, the third **if**, is executed. The Boolean expression here is

```
y > z
```

This is TRUE, so the **then** statement is executed. The net result is that the word "winnow" is printed.

The syntax sketch for the **else if** structure is shown in Figure 4-4. Note that the chain of **if**...**then**...**else if**s can be of any length. Also the final **else** is optional (like all **else**s); this is indicated on the sketch with a dashed box.

The most important thing to remember about the **else if** structure is that no more than one of the statements inside the structure will be performed each time the structure itself is executed. Whenever one of the Boolean expressions evaluates to TRUE, the corresponding **then** statement is executed and no more of the conditions in the chain are tested.

If the final (optional) **else** is present in an **else if** structure, it acts as a catchall: the statement following this **else** is

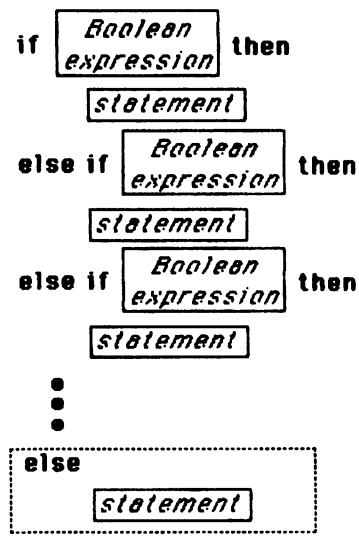


Figure 4-4.

else if structure syntax

executed if none of the previous Boolean expressions is TRUE. If the trailing **else** is present, you can be assured that exactly one of the statements in the **else if** structure will always be performed each time the structure is executed.

For practice in determining how the **else if** structure works, here are three sets of initial values for the variables x, y, and z. For each set, predict the output from the if_lab program; then plug the values into the program and see if you were correct.

```
x := 1.2;
y := 3.4;
z := 1.2
-------
x := 1.2;
y := 0.9;
z := 3.4
-------
x := 1.2;
y := 0.9;
z := 1.0;
```

The **else if** structure allows your program to perform a number of different actions depending on different, mutually exclusive conditions in your program. For example, you might want your program to print out the largest of three numbers. In pseudo-code, you might devise a strategy as follows:

```
get three numbers
if the first number is largest
    print the first number
if the second number is largest
    print the second number
if the third number is largest
    print the third number
```

You should find this easy to translate into Pascal:

```
program max3;
{ find largest of three numbers }

 var
  x, y, z : real;
```

```
begin { max3 }
 write('Enter the first number:');
 readln(x);
 write('Enter the second number:');
 readln(y);
 write('Enter the third number:');
 readln(z);
 write('The largest is ');
 if (x >= y) and (x >= z) then
  writeln(x : 16);
 if (y >= x) and (y >= z) then
  writeln(y : 16);
 if (z >= x) and (z >= y) then
  writeln(z : 16)
end.
```

Type in this program we've called max3 and run it. Verify that no matter what three numbers you type in, the largest one is always printed. (Why does this program use the ">=" relational operator and not ">"? A related question: What happens if two or three of the numbers you enter have the same value?)

This program is fine—it works, anyway—but it can be improved. Notice that if the first number, x, is the largest, you really don't have to check to see if either y or z is the largest. And, conversely, if you know that neither x nor y is the largest number, you don't have to do any further checking at all; by the inexorable rules of logic, z must be the largest number.

So an **else if** structure is really more appropriate for this program. Based on this realization, the program then becomes

```
program max3;
{ find largest of three numbers }

 var
  x, y, z : real;

begin { max3 }
 write('Enter the first number:');
 readln(x);
 write('Enter the second number:');
 readln(y);
 write('Enter the third number:');
```

```
 readln(z);
 write('The largest is ');
 if (x >= y) and (x >= z) then
  writeln(x : 16)
 else if (y >= x) and (y >= z) then
  writeln(y : 16)
 else
  writeln(z : 16)
end.
```

From outside appearances, this program works much like the previous one. This version is cleaner, however; it makes fewer unnecessary tests and it is still clearly correct.

A third version of this program might use a different strategy:

```
program max3;
{ find largest of three numbers }

var
 x, y, z : real;

begin { max3 }
 write('Enter the first number:');
 readln(x);
 write('Enter the second number:');
 readln(y);
 write('Enter the third number:');
 readln(z);
 write('The largest is ');
 if x >= y then
  if x >= z then
   writeln(x : 16)
  else
   writeln(z : 16)
 else if y >= z then
  writeln(y : 16)
 else
  writeln(z : 16)
end.
```

Try this version; does it work? Compare this version of the program with the previous version. Why might this be considered a better solution? (How many comparisons of x, y, and z

does each version make?) Why might this version be considered inferior to the previous version? (If you weren't told the outcome of either version, how long would it take you to figure it out by reading the programs?)

We might also modify Newton once more to take advantage of the **else if** structure. Based on the value of x entered, we can split the program into three exclusive cases: (1) If the input number is positive, calculate the square root as always. (2) If the input number is 0, print the answer as 0. (3) If the input number is negative, print an error message as before.

Retrieve Newton, saving the max3 program first if you want. Make the minor modification to make Newton work as just described:

```
program Newton;
{ Calculate square roots by Newton's method }

 var
  x, old_guess, new_guess : real;

begin { Newton }
 write('Enter a number:');
 readln(x);
 if x > 0.0 then
  begin
   new_guess := x / 2;
   repeat
    old_guess := new_guess;
    new_guess := (old_guess + x / old_guess) / 2
   until new_guess = old_guess;
   writeln('The square root of ', x : 16, ' is ', new_guess : 16)
  end
 else if x = 0.0 then
  writeln('The square root of ', x : 16, ' is ', x : 16)
 else
  writeln('Sorry, this program only works for non-negative numbers.')
end.
```

Verify that the program now has three different outcomes depending on whether you type in a number of positive, zero, or negative value.

As a final example, let's design a simple number-guessing game. The computer will think of a target number, and the computer's user (you, in this case) will attempt to guess it.

This program needs a looping structure to perform the action of asking the user for a guess again and again. Of the looping structures available to us, we can discard the **for** loop right away, since we don't know how many times the program will need to ask for a guess. So our choice is between a **repeat** and a **while**. Remember that a **repeat** is most often useful when an action must be done at least once; a **while** is usually indicated when an action might not be done at all. In this case, the program will always ask for a number at least once, so a **repeat** is the control structure of choice.

Start with a crude strategy expressed in pseudo-code:

```
think of a number
repeat
    get guess from user
until guess is correct
```

To be fair to the person playing this game, the program should say whether an incorrect guess is under or over the correct answer. So a refined algorithm might go like this:

```
think of a target number
repeat
    ask for guess from user
    if guess < target number
        tell user that guess was too low
    else if guess > target number
        tell user that guess was too high
    else
        aha! user guessed correctly
until guess is correct
```

This algorithm is close enough to Pascal so we can translate it directly:

```
program guessing_game;
{ Guess a number thought of by Macintosh }

 const
  MAXGUESS = 100;

 var
  target, guess : integer;
  success : Boolean;
```

```
begin { guessing_game }
 target := random mod MAXGUESS + 1;
 writeln('I am thinking of a number between 1 and ', MAXGUESS : 1);
 success := FALSE;
 repeat
  write('Your guess?:');
  readln(guess);
  if guess < target then
   writeln('Sorry: ', guess : 1, ' is too low')
  else if guess > target then
   writeln('Sorry: ', guess : 1, ' is too high')
  else { got it ! }
   success := TRUE
 until success;
 writeln(guess : 1, ' is correct!')
end.
```

Here we have again used a feature of Pascal not yet discussed. The statement

```
target := random mod MAXGUESS + 1
```

contains the word "random," which looks like an undeclared variable. "Random" is discussed in Chapter 7; for now, consider it to be a predefined integer variable that takes on unpredictable values each time it is examined. Taking this unpredictable value of **mod** MAXGUESS will give us a (still unpredictable) value between 0 and MAXGUESS − 1; adding one to this gives a number between 1 and MAXGUESS.

Play this simple game as long as you like. If you want, try other values of MAXGUESS; make this number smaller for a shorter game and larger for a longer one.

THE case STATEMENT

Suppose that you wanted to write a program that, when given the number for the month (4 for April, 9 for September, and so on), printed the number of days in that month. A good place to start in the design of such a program is to ask yourself: how would I do this in real life? Perhaps you have memorized the verse that follows.

```
Thirty days hath September,
April, June, and November;
All the rest have thirty-one,
Excepting February alone
And that has twenty-eight days clear
And twenty-nine in each leap year.
```

If we were to describe this algorithm in pseudo-code, we might write:

```
if month is 9, 4, 6, or 11 (September, April, June, November)
   days := 30
else if month is not 2 (not February)
   days := 31
else if it is a leap year (month is February)
   days := 29
else
   days := 28
```

You already know how to implement this pseudo-code in Pascal with an **else if** structure. Let us therefore write an entire program based around this design:

```
program days_in_month;
{ calculate number of days in a month }

 var
  month, days, yr : integer;

begin { days_in_month }
 write('Enter the month number:');
 readln(month);
 if (month = 9) or (month = 4) or (month = 6) or (month = 11) then
  days := 30
 else if month <> 2 then
  days := 31
 else { it's February }
  begin
   write('Enter the year:');
   readln(yr);
   if (yr mod 4 = 0) and (yr mod 100 <> 0) or (yr mod 400 = 0) then
    days := 29
   else
```

```
    days := 28
  end;
  writeln('There are ', days : 1, ' days in the month.')
end.
```

The test in this program for a leap year is complex, but correct. According to the Gregorian calendar, if the year is divisible by 4 (year **mod** 4 is 0), the year is a leap year unless it is also divisible by 100 and not divisible by 400. (So 1900 and 1800 were not leap years, but the year 2000 will be.) Evaluate the complex Boolean expression by hand using a few different values for the year.

Another type of statement allows a slightly different way of doing the same thing: the **case** statement. The **case** statement lets you choose between many cases of the possible values of a variable. This program could be rewritten with a **case** as follows:

```
program days_in_month;
{ calculate number of days in a month }
var
 month, days, yr : integer;

begin { days_in_month }
 write('Enter the month number:');
 readln(month);
 case month of
  9, 4, 6, 11 :
   days := 30;
  1, 3, 5, 7, 8, 10, 12 :
   days := 31;
  2 :
   begin
    write('Enter the year:');
    readln(yr);
    if (yr mod 4 = 0) and (yr mod 100 <> 0) or (yr mod 400 = 0)
     then days := 29
    else
     days := 28
   end
 end;
 writeln('There are ', days : 1, ' days in the month.')
end.
```

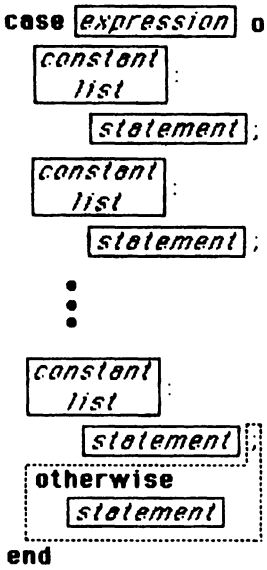


Figure 4-5.

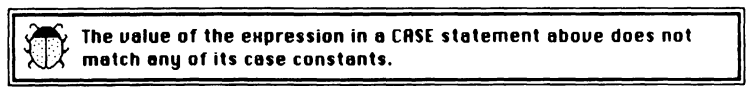
case statement syntax

Essentially this **case** statement says, "Look at the value of the variable month; if it is 9, 4, 6, or 11, assign the value 30 to the variable days. If month has the value 1, 3, 5, 7, 8, 10, or 12 (the other months except February), assign the value 31 to days. Finally, if month has the value 2, ask for the year. If the year is a leap year, then days is 29; otherwise, it's 28." Type in this program and verify that it correctly calculates the number of days in any month.

The general syntax of the **case** statement is shown in Figure 4-5. The expression used between the words **case** and **of** may be of any type we've discussed so far except real. (As we introduce other types, we will discuss whether they may be used in a **case** statement.) As shown in the example, a *constant list* is what you would expect: one or more constant values separated by commas. Each constant in the constant list must be of the same type as the **case** expression. Any given constant value may appear in all the constant lists a total of once. Although the syntax sketch doesn't show it, Pascal permits a semicolon preceding the **case**'s final **end**.

The **otherwise** clause shown at the end of the **case** syntax sketch is optional. If it is present, the semicolon preceding the word **otherwise** is also necessary. Like the final **else** in all **else if** structures, the **otherwise** clause is a catchall; if the value of the **case** expression doesn't match any of the constants in any of the lists, the statement following **otherwise** executed.

To see why an **otherwise** escape clause is often useful, run the program and enter a value of 13 for the month.



The **otherwise** allows an easy alternative to this sort of impolite behavior. Modify the end of the program slightly:

```
program days_in_month;
{ calculate number of days in a month }
 var
  month, days, yr : integer;

begin { days_in_month }
 write('Enter the month number:');
```

```
readln(month);
case month of
 9, 4, 6, 11 :
  days := 30;
 1, 3, 5, 7, 8, 10, 12 :
  days := 31;
 2 :
  begin
   write('Enter the year:');
   readln(yr);
   if (yr mod 4 = 0) and (yr mod 100 <> 0) or (yr mod 400 = 0)
    then days := 29
   else
    days := 28
  end;
 otherwise
  days := 0;
end;
if days <> 0 then
 writeln('There are ', days : 1, ' days in the month.')
else
 writeln('Sorry: invalid month.')
end.
```

Rerun the program; this time the unexpected input is handled in a more civil manner. (Verify that normal input is handled as before.)

Whether you use an **else if** structure or a **case** statement in a given program is sometimes a matter of taste. The **else if** is more general: it can handle many different kinds of alternative decisions, as we have seen. The **case** statement, on the other hand, is restricted to performing different actions based on different values of a single expression.

Anything that can be done with a **case** can also be accomplished with an **else if** structure. However, a **case** statement will often run faster and take up less memory than the equivalent **else if** structure. A **case** statement can also be more clear and concise to someone reading your program.

A final caveat: the **otherwise** clause is not present in Standard Pascal. If you mean to run the programs you write using versions of Pascal other than the Macintosh variety, avoid **otherwise**; it can make your programs more difficult than necessary to translate to another version of Pascal. (In the jargon, using **otherwise** makes your program less

portable since it can't easily be moved to a different system.) A portable way to avoid the problem of unexpected input values is to protect the **case** statement with an **if** test. In our program, this could be done as follows:

```
•
•
•
if (month >= 1) and (month <= 12) then
 case month of
  •
  •
  •
```

THE goto STATEMENT

In this section we will discuss a part of the Pascal language you will seldom want to use: the **goto** statement. A **goto** statement in your program commands that the flow of control jump to another part of the program.

To use a **goto**, you must specify the statement you want your program to go to. This is accomplished by prefixing this statement with a *label.* Any statement in your program may be labeled, including empty statements. A label is any number in the range 0-9999.

All labels that exist in your program must be declared, just as variables and constant identifiers must be declared. Labels are declared in the *label definition part* of your program; the syntax of the label definition part is shown in Figure 4-6. To declare labels, just use the reserved word **label**, followed by a *label list:* one or more labels separated by commas.

The label definition part in a program goes before the constant definition part and after the **program** line. So we have an addition to the Pascal program syntax; the new part is shown in its proper place in Figure 4-7.

Figure 4-6.

Label definition part syntax

Once you define a label you can use it to label a statement; this syntax is shown in Figure 4-8. The label and a colon precede the statement you want to label. A labeled statement may be the target of a **goto** statement; the **goto** statement's syntax is shown in Figure 4-9.

Macintosh Pascal will consider it an error if you try to go

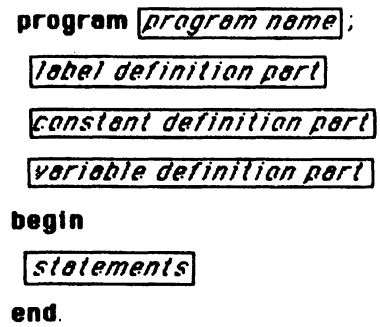


Figure 4-7.

Pascal program syntax (still incomplete)

to a nonexistent label, try to label a statement without declaring the label first, or try to have two or more statements in your program with the same label.

For a specific example of how labels and **gotos** are used, type in the following goto_lab program, which admittedly doesn't do much. As always, try to guess the output before you run the program.

```
program goto_lab;
{ experiments with goto }

 label
  1, 2, 3, 4, 5, 6;

 var
  i, j : integer;

begin { goto_lab }
 j := 0;
1 :
 for i := 1 to 10 do
  begin
2 :
  write('a');
  if j = 1 then
   goto 4;
  if i > 1 then
   goto 3;
```

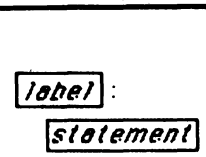


Figure 4-8.

Labeled statement syntax

Figure 4-9.

goto statement syntax

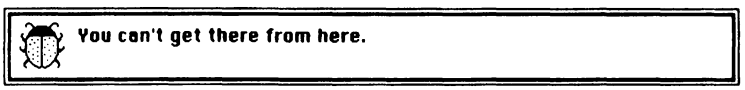
```
    write('b');
    goto 5;
  3:
    if j >= 2 then
     goto 6;
    write('c');
    j := j + 1;
    goto 2;
  4:
    write('d');
    j := j + 1;
    goto 1;
  5:
    write('r')
   end;
  6:
  end.
```

Note that it is possible to use a **goto** to jump from inside a statement to outside it. In this program, the "**goto** 6" and "**goto** 1" statements jump out of the **for** loop. It is also possible to jump to statements at the same nesting level—statements "**goto** 5" and "**goto** 2," for example.

It is illegal, however, to use **goto** to try to go to the inside of a statement from the outside. For example, insert the statement

```
goto 2
```

just after the initial **begin** of the program. Run the program and you'll get this amusing bug box:



The goto_lab program is intentionally convoluted and hard to understand; it demonstrates most of the problems resulting from the unrestricted use of the **goto** statement. All of the other statements in Pascal are more or less self-explanatory; when you see a **for** statement, for example, it is easy to determine such details as what statements are executed within the loop and how often the loop is performed.

A **goto** statement is almost never that clear. To determine the action of the **goto**, you must first figure out the conditions under which the **goto** statement will be executed. You must then search through the program for the specified label and unravel what the program does when control is tranferred there. In perusing someone else's program (or even one of your own) you will often find the reasons behind the use of a **goto** completely mysterious.

Related problems also arise from the use of multiple labels in a program. When there is more than one way for control to pass to a statement, it is difficult to determine how many times and under what conditions that statement will be executed. This can be nightmarish when you are debugging a program and trying to work backward from that statement.

So there are many reasons not to use the Pascal **goto**. A general guideline is to use it only as a last resort, when all the more restricted control structures we have discussed have been tried and found wanting. For example, you might justifiably use **goto** to escape from a deeply nested set of statements when your program detects some sort of serious error condition. All such exceptional situations should be clearly commented so that anyone reading the program will be able to follow the logic involved.

MACINTOSH PASCAL DEBUGGING AIDS 5

Debugging is not an activity relished by the average programmer; in fact, it is usually considered the most frustrating and nerve-wracking aspect of writing a program.

—*Edward Yourdon*
Techniques of Program Structure and Design
(Prentice-Hall, 1975)

Until now, much of our discussion has dwelled on how your programs might *not* work. This may have seemed a strange emphasis to you, especially if you have not had a lot of prior experience writing programs. As you gain that experience, however, you will find that bugs are inevitable in your programs. You will come to appreciate Murphy's Law ("If anything can go wrong, it will").

You have already experienced the species of bug known as the syntax error. Syntax errors occur when your program fails to conform to the rules of Pascal. Examples of common syntax errors include missing or extra semicolons, misspelled words ("intger" instead of "integer," for example), unbalanced parentheses, and undeclared variables.

You have also encountered *run-time* errors, yet another type of bug. A program with a run-time error will begin running (hence the name), but some condition that develops when the program runs will cause the program to terminate prematurely with an error message. Examples of run-time

errors include overflow and division by zero. Macintosh Pascal also considers many instances of illegal type mixing (discussed in Chapter 2) to be run-time errors.

A third type of bug—the most insidious—is the *logic error.* Such errors generate no error messages; in many cases, a program with a logic error may even appear to run properly. Logic errors can be as simple as writing a plus sign where you should have written a minus sign. Since computers have no way of knowing what you really meant to do, a syntactically correct logic error that generates no run-time errors cannot be detected by the computer; you must roll up your sleeves and do it yourself.

Program bugs aren't anything to be ashamed of, or frightened of either. As you become a better programmer, you will also get better at debugging. (Your bugs will also get harder to find and fix, but that's another story.)

Another reason we have emphasized bugs and their elimination is that Macintosh Pascal provides an exceptionally good debugging environment. When Macintosh Pascal detects a problem within your program, its error messages (with few exceptions) are accurate and in understandable language with minimal jargon. The magic hand, at the left side of the program window usually shows the approximate location of the error. And best of all, fixing a program error is usually accomplished using only a few mouse movements, clicks, and keystrokes. Other versions of Pascal and other languages aren't so kind, as you may know.

In this chapter we will concentrate on additional features of Macintosh Pascal that make debugging your programs easier. For the purposes of this chapter, we will develop a program containing many types of bugs. Then we will show how to exterminate each one with the help of the debugging aids provided by Macintosh Pascal.

CHECKING PROGRAM SYNTAX

Let's start by creating a program that will find a solution to the equation "$x^2 + x - 1 = 0$". The positive solution to this equation is known as the *golden ratio* because geometric figures based on this ratio are considered to be pleasing to the eye.

A note to readers who are not mathematically inclined: you should probably only skim the following explanation and

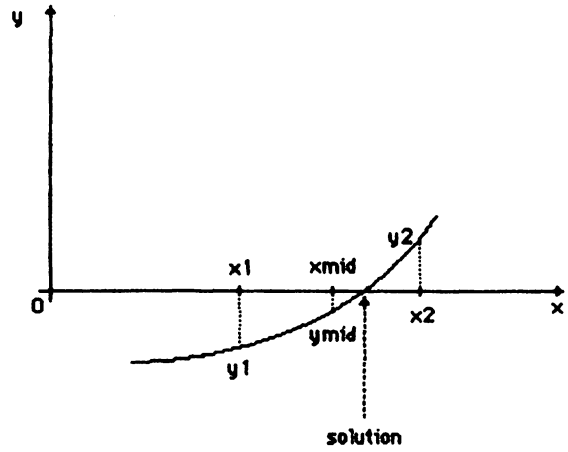


Figure 5-1.

The method of bisection

concentrate on the mechanics of the debugging method instead.

The method our program will use to find the golden ratio is the *method of bisection*, illustrated in Figure 5-1. The curved line shows roughly how the equation $x^2 + x - 1$ changes as the value of x changes. The value of x for which the curved line crosses the x axis is the solution we are seeking.

The method of bisection assumes you are given two values of x that straddle the actual solution; these values are denoted as x1 and x2 in Figure 5-1. Then perform these steps:

1. Compute $x^2 + x - 1$ for each of these values of x; call the results y1 and y2, respectively. Note that, by our assumptions, one of these values will be positive, the other negative.
2. Now pick a value of x halfway between x1 and x2 (denoted as xmid in Figure 5-1) and calculate $x^2 + x - 1$ for that value; call this result ymid.
3. If the sign of ymid is different from the sign of y2, the solution lies between xmid and x2. (This is the case in Figure 5-1.) Set x1 to the value of xmid, y1 to the value of ymid, and return to Step 2.

4. If the sign of ymid is different from the sign of y1, the solution lies between x1 and xmid. Set x2 to the value of xmid, y2 to the value of ymid, and return to Step 2.
5. The only other possibility is that ymid is 0, in which case we've found the solution.

Informally, we can see that each time Step 3 or Step 4 is done, the values of x1 and x2 will get closer to one another. Eventually, we might suppose, the values will close in on the correct answer, and since the computer is doing all the work, we might also suppose we don't have to worry about the details of the process or how long it will take. (As we'll see, these suppositions are incorrect.) A pseudo-code for this algorithm might look as follows:

```
get values for x1 and x2
    (lower and upper bounds for solution)
calculate y1 and y2 from x1 and x2
while solution hasn't yet been found
    calculate xmid halfway between x1 and x2
    calculate ymid from xmid
    if ymid and y2 have opposite signs
        set x1 to xmid
        set y1 to ymid
    else if y1 and ymid have opposite signs
        set x2 to xmid
        set y2 to ymid
    else
        ymid must be zero and so xmid is the solution
end while
output xmid as solution
```

A rough translation of this pseudo-code into Pascal follows. Type in this program *exactly as shown.* (Yes, it contains errors.)

```
program golden
{ find the golden ratio! }

 var
  xnid, x1, x2 : real;
  ymid, y1, y2 : real;

begin { golden }
 write('Enter lower bound for solution:');
```

```
readln(x1);
write('Enter upper bound for solution:');
readln(x2);
y1 := x1 * x1 + x1 - 1.0;
y2 := x2 * x2 + x2 - 1.0;
while ymid <> 0.0 do
 begin
  xmid := (x1 + x2) div 2.0;
  ymid := xmid * xmid + xmid - 1.0;
  if y1 * ymid < 0.0 then
   begin
    x1 := xmid;
    y1 := ymid
   end
  else if ymid * y2 < 0.0 then
   x2 := xmid;
  y2 := ymid
 end
end;
writeln('Solution is ', xmid : 16)
end.
```

(Before we proceed, note that this program uses an easy method to determine whether two numbers differ in sign. Simply multiply the two numbers; if the product is less than 0, their signs are different.)

As you know, any syntax error will prevent your program from running at all; you must fix all program syntax errors before you can proceed. Macintosh Pascal provides an easy way you can check your program for syntax errors without actually running it: choose the Check option from the Run menu. (As a shortcut, press the COMMAND-K combination on the keyboard.)

Use the Check option to find the syntax errors in this program; as each one is detected, fix it. Here is a list of the causes of the errors, but try to find them on your own first:

- The **program** line is missing a semicolon at its end.
- The variable "xmid" is misspelled "xnid" in the **var** section.
- The second **if...then** is missing a **begin** after the **then**.

Once you have fixed these three errors, use the Check option to verify that there are no more syntax errors in your program. (When a program is syntactically correct, the

Check option doesn't appear to do anything, which makes a certain amount of sense: no news is good news.)

You can use the Check option periodically while you are entering a long program so you can detect syntax errors soon after they are made. If you do this, be sure you can distinguish between real syntax errors and those arising because your program is incomplete.

STEPPING THROUGH YOUR PROGRAM

After you fix the syntax errors found by the Check option, your program should look as follows:

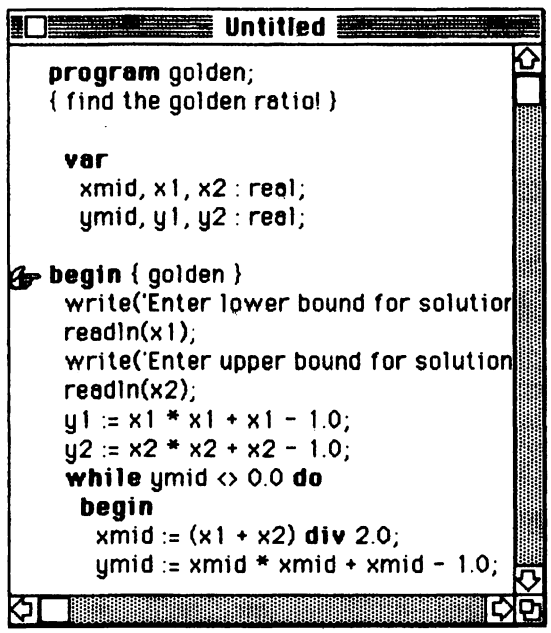
```
program golden;
{ find the golden ratio! }

 var
  xmid, x1, x2 : real;
  ymid, y1, y2 : real;

begin { golden }
 write('Enter lower bound for solution:');
 readln(x1);
 write('Enter upper bound for solution:');
 readln(x2);
 y1 := x1 * x1 + x1 - 1.0;
 y2 := x2 * x2 + x2 - 1.0;
 while ymid <> 0.0 do
  begin
   xmid := (x1 + x2) div 2.0;
   ymid := xmid * xmid + xmid - 1.0;
   if y1 * ymid < 0.0 then
    begin
     x1 := xmid;
     y1 := ymid
    end
   else if ymid * y2 < 0.0 then
    begin
     x2 := xmid;
     y2 := ymid
    end
  end;
 writeln('Solution is ', xmid : 16)
end.
```

Run this program by choosing Go from the Run menu, as usual. Enter a value of 0.0 for the lower bound and 1.0 for the upper bound when requested. The program will appear to work, generating the solution:

```
Solution is 0.0000000e+0
```

Unfortunately, this answer is incorrect, as you can verify by plugging it into the original equation. You'll remember we called this species of bug a logic error. The program ran to completion but failed to give the right answer.

To find out what happened, let's try out a couple of Macintosh Pascal's debugging tools: the *Step* and *Step-Step* options in the Run menu. First, choose the Step option. (Shortcut: press COMMAND-S.) The Program window should appear as follows:



The magic hand in the left side of the Program window is now a pointing finger; you saw this finger in Chapter 2 when you stopped a runaway program using the Halt option in the Pause menu. The pointing finger always indicates the line of the program Macintosh Pascal is about to execute. In this case, the program hasn't quite started up yet; the finger points to the initial **begin**.

Choose Step (or press COMMAND-S) once more. The finger moves down one line to the first **write** statement. Choose Step again, and the finger moves again, this time to the first **readln** statement. Notice that the write command was actually executed; the prompt "Enter lower bound for solution:" should appear in the Text window.

Don't type anything in response to the prompt because the **readln** statement hasn't begun to execute yet. To execute the **readln** statement, choose Step again. You should see the blinking insertion point appear in the Text window, showing that the Macintosh now expects you to type something in response to the prompt. As before, enter a value of 0.0 as the lower bound.

Continue stepping through the program, entering a value of 1.0 for the upper bound of the root when the second **readln** is executed. When the finger reaches the **while** statement, choose Step once more. Notice that the finger jumps all the way down to the **writeln** statement. This means the body of the **while** loop is never executed, which is the source of the current problem. Continue stepping until the program stops.

There is an alternative to continually choosing the Step option (or continually pressing COMMAND-S). The Step-Step option in the Run menu is an automatically repeating Step; instead of stopping the program on each line, Step-Step merely pauses a moment and then continues. The net effect is that your program is run in very slow motion; you can easily observe the control flow by watching the magic hand. Try executing the program with Step-Step.

We have found our logic error (the first one, anyway). The Boolean expression for the **while** loop is

```
ymid <> 0.0
```

This compares the value of the variable ymid against 0. But the program hasn't explicitly set ymid to any value when this expression is first evaluated. To evaluate the Boolean expression, therefore, Macintosh Pascal must assume some initial value for ymid; this assumed value is, unfortunately, 0.

Our mistake was in trying to ignore the following rule:

Never use the value of an variable until that variable has been given a value.

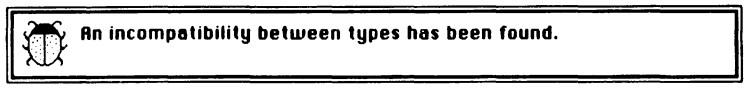
In computer jargon, variables should be *initialized* before their values are used. Some versions of Pascal (including, apparently, Macintosh Pascal) set all variables to 0 (or equivalent null values) before a program is run. Assuming that a variable is automatically initialized to any particular value, however, is sloppy programming style.

For this particular program, perhaps the best solution is to turn the **while** loop into a **repeat** loop, as follows:

```
program golden;
{ find the golden ratio! }

 var
  xmid, x1, x2 : real;
  ymid, y1, y2 : real;

begin { golden }
 write('Enter lower bound for solution:');
 readln(x1);
 write('Enter upper bound for solution:');
 readln(x2);
 y1 := x1 * x1 + x1 - 1.0;
 y2 := x2 * x2 + x2 - 1.0;
 repeat
  xmid := (x1 + x2) div 2.0;
  ymid := xmid * xmid + xmid - 1.0;
  if y1 * ymid < 0.0 then
   begin
    x1 := xmid;
    y1 := ymid
   end
  else if ymid * y2 < 0.0 then
   begin
    x2 := xmid;
    y2 := ymid
   end
 until ymid = 0.0;
 writeln('Solution is ', xmid : 16)
end.
```

Note that the Boolean expression is also turned around: the loop is run until ymid becomes 0. Run this version of the program, once more entering 0.0 and 1.0 for the lower and upper

bounds for the solution. This time the result is a run-time error:



The magic hand shows thumbs-down on the statement

```
xmid := (x1 + x2) div 2.0
```

A moment's thought should give you the solution to this error: the **div** operator is used for integer division, and we are using it for real division. Some versions of Pascal would treat this as a syntax error. Macintosh Pascal, however, delays its checking on type mixing until run-time. To exterminate this bug, replace **div** with a slash:

```
xmid := (x1 + x2) / 2.0
```

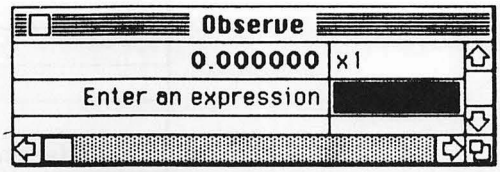
THE OBSERVE WINDOW

We have eliminated some bugs from our program, but some subtle ones remain. Run the program again with the usual inputs of 0.0 and 1.0 in response to the two prompts.

This time you should observe that nothing happens after you type the numbers. The program is stuck in an infinite loop. (How can you tell the difference between a program with an infinite loop and a program that just takes a long time to get the answers? This is a very tricky question with no concrete answer. For now, simply wait for as long as you need to be convinced that the program will never do anything on its own.) To stop the program, choose the Halt option from the Pause menu.

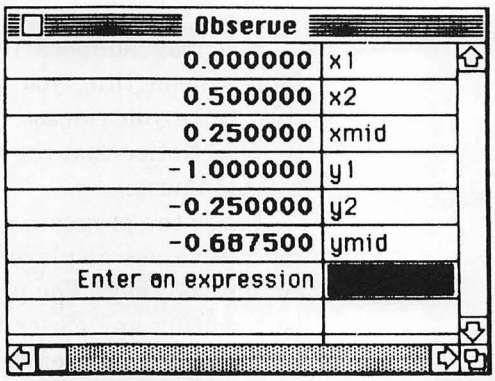
Let's use another debugging tool provided by Macintosh Pascal to try to discover the cause of this logic error. Choose

the Observe option from the Windows menu to display the Observe window:

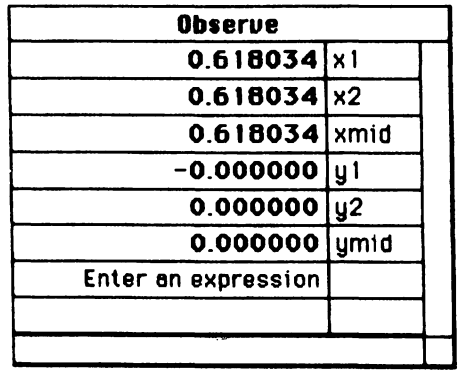


To use the Observe window, one enters expressions into the boxes on the right side of the window. Note that there is a blinking insertion point across from the request to "Enter an expression." All the editing rules you learned for entering program text also apply in the Observe window; you may, if you want, even cut, copy, and paste. When the program is temporarily halted as it is now, these expressions are evaluated and their values are displayed in the corresponding boxes on the left.

To see how it works, type "x1" and press the ENTER key (note that you use the ENTER key, not the RETURN key). After a bit of disk whirring the result should look like this:



Remember that the name of a single variable is a perfectly good expression. This tells us the current value of x1 is 0. (Is this enough of a clue to tell you what the problem is?) Let's display the values of our other variables; enter the identifiers "x2", "xmid", "y1", "y2", and "ymid", pressing the ENTER key after each one. Expand the Observe window so you can see all six values simultaneously. (Also move the Observe window so you can see the Text window.) Now can you figure out the bug? The Observe window should now look like this.



Our program is stopped, but it can be started up again from where it left off. In general, whenever you halt a program, Macintosh Pascal remembers the values of all the variables and where the program stopped. When you choose the Go option, the program starts up from the point where it was halted.

So start the program again by choosing Go from the Run menu (or pressing COMMAND-G). Once more the program is in an infinite loop; nothing is displayed in the Text window. Simply running the program will not change the Observe window either; the Observe window is only modified (or *updated*) when the program is halted.

You may temporarily halt the program by pressing the Pause menu title; you need not select the Halt option. Try this. Once you release the Mouse button, the program continues. (Notice that the Observe window is updated after you release Pause.)

Pause the program a number of times, each time examining the values displayed in the Observe window. They don't change, do they? The problem seems to be that the program isn't getting any closer to a solution.

Why not? Remember that the assumptions behind the bisection method were that the values of y1 and y2 were of opposite sign. The values for y1 and y2 shown in the Observe window are both negative. Compare the program with the pseudo-code; where did we go wrong?

The problem is that we muffed the translation from pseudo-code to Pascal. The pseudo-code says

```
if ymid and y2 have opposite signs
    set x1 to xmid
    set y1 to ymid
else if y1 and ymid have opposite signs
```

```
set x2 to xmid
set y2 to ymid
```

But the program says

```
if y1 * ymid < 0.0 then
 begin
  x1 := xmid;
  y1 := ymid
 end
else if ymid * y2 < 0.0 then
 begin
  x2 := xmid;
  y2 := ymid
 end
```

The first time through the loop, the program sets x2 to 0.5 instead of setting x1 to 0.5. From that point on, all values of y1, y2, and ymid are negative, so no changes to x1 or x2 are made again. (Work through the program, if you want, to see this for yourself.)

Halt the program and change it to more accurately reflect the pseudo-code. (Remember you will have to click in the Program window first to make it active.) Your program now reads as follows:

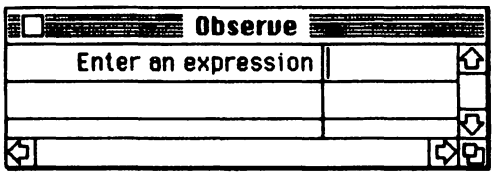
```
if y1 * ymid < 0.0 then
 begin
  x2 := xmid;
  y2 := ymid
 end
else if ymid * y2 < 0.0 then
 begin
  x1 := xmid;
  y1 := ymid
 end
```

Note that changing the program removes the pointing finger from the Program window. Macintosh Pascal resets your program when you make changes to it, so you may no longer start your program from where it stopped. You may also reset your program by choosing the Reset option from the Run menu.

Try once more to run your program, again entering 0.0 and 1.0 for the bounds. There is something else wrong. The program is still stuck in an infinite loop.

Halt the program; you should still be able to see the values

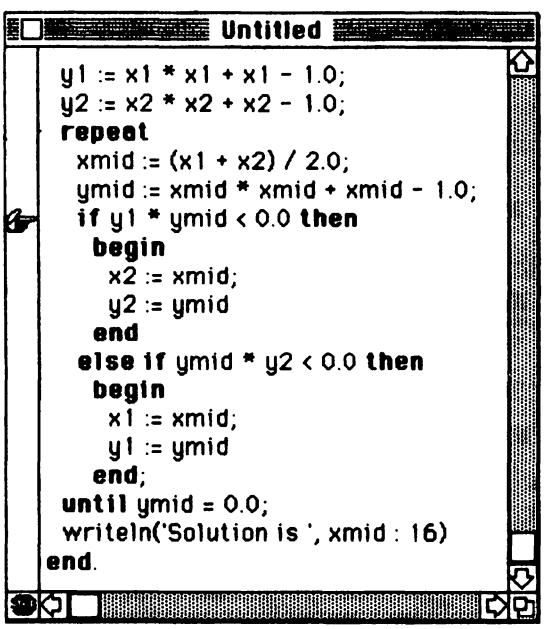
in the Observe window, and they should look like this:



At least this is different from what we saw before. And we are clearly getting closer to the right answer: the x values are all the same, indicating that the process is converging on a solution. The y values are all 0, indicating that the solution is correct. But why isn't the program giving us the right answer?

STOPS

Let's use yet another debugging tool to get a better idea of what's going on. Choose the Stops In option from the Run menu. You will then see a change in the Program window:

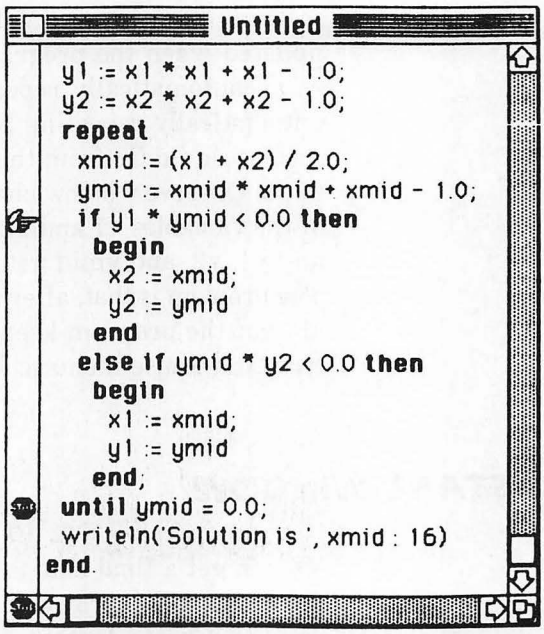


(The magic hand may well point to a different line from that shown here.) When you choose Stops In, the Program window develops a bar on its left side, similar in size to the scroll bar on the right side. We'll refer to this bar as the *stop bar.* A tiny stop sign appears in the lower-left corner of the window.

Move the pointer over to the stop bar. You should notice that when the pointer is in the stop bar it appears as a stop sign. Move the stop-sign pointer across from the line:

```
until ymid = 0.0
```

(Scroll the Program window if necessary.) Click the Mouse button. Note that this acts to set a stop sign at the desired line:



Appropriately enough, a stop sign acts to stop the program when the line across from the sign is about to be executed. You may set as many stop signs in your program as you want. (A stop must be in an executable part of your program, though, between **begin** and **end.**) If you want to remove a stop sign that you have put in the program, just click it.

Remember that our problem is that our program is stuck in the **repeat...until** loop. So we have chosen a critical point to insert the Stop: the place where the loop exit condition is

tested. To see how it works, first choose Reset from the Run menu and then choose Go. As always, enter 0.0 and 1.0 in response to the prompts. The next thing you should notice is that the program halts with the pointing finger at the stop sign you placed at the **until**. Also note that the values in the Observe window are updated when the program halts, as usual.

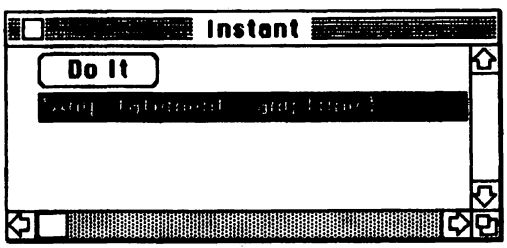
You may restart the program by choosing Go again. When the program returns to the **until** line it will halt once more. Choose Go a few more times; each time you should note that the values in the Observe window are updated when the program hits the stop sign. (Since we are debugging the program you should also notice how the values in the Observe window change.)

The only option in the Run menu we have not yet discussed is *Go-Go*. Go-Go works like Go, except that when the program encounters a stop sign, the program only pauses momentarily instead of stopping. Values in the Observe window are updated when the program pauses. (You can think of Go-Go as an automatically repeating Go, just as Step-Step was an automatically repeating Step.)

Choose Go-Go from the Run menu and see how the values in the Observe window change. Note especially that the values of the variables x1 and x2 get closer and closer to each other, and y1, y2, and ymid get closer and closer to 0, as expected. The problem is that, after a time, the variables don't change at all, but the program keeps running. Once again, choose Halt from the Pause menu to stop the infinite loop.

THE INSTANT WINDOW

We can get a final clue to our program bug by using another debugging tool provided by Macintosh Pascal: the *Instant window*. Choose Instant from the Windows menu to display the Instant window:



As the comment indicates, one uses the Instant window to perform "any statements, any time." Try it; enter the following statement in the Instant window. (Typing and editing in the Instant window works just as it does in the Program window.)

```
writeln(ymid : 16)
```

After you have entered this line, click the Do It button. After some disk whirring, the program will display the value 3.6705515e−8 in the Text window, just as if the line you typed were in your program.

Try placing the following statements in the Instant window and executing them:

```
writeln('x2 - x1 is ', x2 - x1 : 16);
writeln('xmid - x1 is ', xmid - x1 : 16);
writeln('x2 - xmid is ', x2 - xmid : 16)
```

The results show what the problem is with our program:

```
x2 - x1 is 5.9604645e-8
xmid - x1 is 5.9604645e-8
x2 - xmid is 0.0000000e+0
```

The Pascal calculation that is supposed to give a value for xmid halfway between x1 and x2 is failing us. In mathematics, the value of xmid can never equal either x1 or x2. And we assumed that Pascal would behave the same way. Unfortunately, the limited precision of the Pascal real data type makes this assumption false. Pascal has given us a value of xmid precisely equal to x2. The values of x1 and x2 are so close together that, as far as Macintosh Pascal is concerned, there is no number halfway between them.

The logic error we made in this program was in assuming that there was some value of x that would give a y value of precisely 0. Although this is mathematically true, it is false in Pascal, at least in this case.

We need to modify te program slightly to make it work. Instead of looking for a value of x that will result in $x^2 + x - 1$ being precisely 0, we should look for a value that gives us a result very close to 0. Try the following slight modification to the program.

```
program golden;
{ find the golden ratio! }

 const
  EPSILON = 1.0e-6;

 var
  xmid, x1, x2 : real;
  ymid, y1, y2 : real;

begin { golden }
 write('Enter lower bound for solution:');
 readln(x1);
 write('Enter upper bound for solution:');
 readln(x2);
 y1 := x1 * x1 + x1 - 1.0;
 y2 := x2 * x2 + x2 - 1.0;
 repeat
  xmid := (x1 + x2) / 2.0;
  ymid := xmid * xmid + xmid - 1.0;
  if y1 * ymid < 0.0 then
   begin
    x2 := xmid;
    y2 := ymid
   end
  else if ymid * y2 < 0.0 then
   begin
    x1 := xmid;
    y1 := ymid
   end
 until (ymid < EPSILON) and (ymid > -EPSILON);
 writeln('Solution is ', xmid : 16)
end.
```

Before you run this version of the program, remove the stop sign from the left margin by choosing the Stops Out option from the Run menu and close the Observe window. Verify that this modification—at last—makes our program work.

Questions for your further study:

- There is another solution to the equation that lies between −2.0 and 0.0. Can you use this program to find that solution as well?

- What happens when the numbers you enter for lower and upper bounds for the solution do not "straddle" a solution to the equation? (Try entering 10 and 20, for example.) Can you modify the program to first check that the entered values actually do straddle a solution and then print a suitable error message if they don't? What if one of the bounds entered by the user of the program is actually a solution to the equation? Can you make your program handle this situation in a logical manner?
- How would you modify this program to find square roots? (Hint: what is the solution to the equation $x^2 - 2 = 0$?) How could you use it to find cube roots?

In working through this example, you should have learned the mechanics of using the Macintosh Pascal debugging tools. In addition, you should also have obtained an inkling of how to debug your programs in an intelligent manner.

Debugging programs and avoiding bugs in the first place is often a matter of carefully questioning your assumptions. When you write a loop, for example, you will find yourself asking, Under what conditions will this loop be entered? When, if ever, will the program exit from the loop? Is there any way the loop could execute forever? As you gain experience, you will learn to ask yourself the right questions as you are designing and debugging.

MORE DATA TYPES

6

God created the integers; all
the rest is the work of man.

Leopold Kronecker

Computers were first used for numeric calculations of all sorts, so it is not too surprising that we have concentrated on that aspect of Pascal. But as you know, computers are used for much more than working with numbers. For example, word processing programs such as MacWrite deal with text material, while picture processing programs like MacPaint allow you to manipulate images on the screen. Deep inside the computer, however, all numbers, words, and pictures are simply different patterns of electrical voltages and currents in the computer's memory.

Loosely speaking, a Pascal data type provides you with a way of looking at memory: *this* chunk of memory represents a real number, but *that* chunk is an integer, and that chunk over there is a Boolean. You don't need to know—unless you want to—how the underlying nuts and bolts electronics are translated into the more abstract data structures of Pascal.

In this chapter, we will consider six more data types available to your Pascal programs: the *character* data type, the *string* data type, the *long integer* data type, and three additional types of real numbers; these are *extended*, *double*, and *computational*. (Of these new data types, only the character type is present in Standard Pascal.) Each of these new data types—new ways of looking at memory—will give you added flexibility in writing your programs.

CHARACTERS

Character data is represented in Pascal by the *char* data type. To declare character variables, you simply use the word char just as you have already used the words real and integer, for example:

```
var
 ch : char;
 delim : char;
```

This says the two variables ch and delim are characters (or more precisely, ch and delim are variables of the type char).

One writes a *character constant* by enclosing a single character between apostrophes. For example, to set the value of the character variable ch to the letter A, you would write:

```
ch := 'A'
```

Note that it would be illegal to have more than one character between the apostrophes in this case because the value represented cannot by definition be more than a *single* character. You may remember from Chapter 1 that we needed to have a special rule for writing apostrophes within character strings: all apostrophes must be doubled. The same rule applies to character constants; an apostrophe character is represented by writing two apostrophes between two apostrophes. For example, you might write:

```
delim := ''''
```

This assignment statement sets the character variable delim to a *single* apostrophe.

You should think of character assignment statements exactly the same way you think of an assignment like

```
i := 137
```

In both cases, a value is being stored in a variable. The rules you already know concerning assignment statements do not change when you are dealing with characters; you still need a single character variable name on the left side of the := symbol and a character-valued expression on the right.

You may also, of course, assign the value contained in one character variable to another. For example,

```
delim := ch
```

Pascal's general rules against type mixing apply to characters. It is fruitless to attempt arithmetic operations on character values. It is illegal to assign character values to variables of other types or, conversely, to assign expressions of other types to character variables. (By now you know such attempts will result in the "type incompatibility" bug box.)

You may use the write and writeln commands to display the values of character expressions. As with string output, you may specify an optional parameter to right-justify the character in a specified field width. Try the following program. (Expand the Text window before you run the program.)

```
program character_lab;
{ experiments with characters }

 var
  top, leftside, rightside, bottom, stem : char;
  i : integer;

begin { character_lab }
 top := '*';
 leftside := '/';
 rightside := '\';
 bottom := '_';
 stem := '|';
 writeln(top : 11);
 for i := 1 to 10 do
  writeln(leftside : 11 - i, rightside : 2 * i);
 for i := 1 to 20 do
  write(bottom : 1);
 writeln;
 for i := 1 to 3 do
  writeln(stem : 11);
 writeln('Merry Christmas!')
end.
```

After trying this program, and while you are still in the spirit, you might try your hand at a somewhat flashier display. The following program is a fun diversion that continues the theme of the previous program:

```
program tree2;
{ draw a tree }

var
 i, j : integer;

begin { tree2 }
 framerect(180, 95, 195, 105);
 moveto(100, 30);
 lineto(50, 180);
 lineto(150, 180);
 lineto(100, 30);
 lineto(100, 15);
 while TRUE do
  begin
   i := random mod 7 + 1;
   j := random mod i + 1;
   invertcircle(100, 15, i);
   invertcircle(100 - (i - 2 * j + 1) * 6, 30 + 20 * i, 2)
  end
end.
```

Descriptions of the commands used in this program will be given in Chapters 7 and 8. Alternatively, character values can be read from the keyboard using the readln command. Try this program:

```
program character_lab;
{ experiments with characters }

var
 ch : char;

begin { character_lab }
 write('Please enter a character:');
 readln(ch);
 writeln('You entered the character ', ch : 1)
end.
```

This program prompts for and accepts a single character. As usual with readln, you signal when you are done typing by pressing the RETURN key. Do not enclose the input character in apostrophes, however. (If you do, the character read will be an apostrophe. Why?)

You may also read single characters typed at the Macin-

tosh keyboard without waiting for the RETURN key to be typed. This involves using a new command called *read.* Change the program you just tried so that it uses read instead of readln, as follows:

```
program character_lab;
{ experiments with characters } ·

 var
  ch : char;

begin { character_lab }
 write('Please type a character:');
 read(ch);
 writeln;
 writeln('You typed the character ', ch : 1)
end.
```

Run the program again to verify that it doesn't wait for you to press the RETURN key; the program continues after you type a single character. Why is there an extra writeln in the program? What happens if you leave it out? (Other versions of Pascal may wait for a RETURN before continuing even when you use read instead of readln.)

You may also compare character values using relational operators. The simplest kind of comparison involves checking for equality. To see how it works, modify your program to ask for a specific input and respond accordingly:

```
program character_lab;
{ experiments with characters }

 var
  ch : char;

begin { character_lab }
 writeln('Do you enjoy programming?');
 write('Please type y for yes, n for no:');
 read(ch);
 writeln;
 if ch = 'y' then
  writeln('Ah, I''m glad to hear it.')
 else
  writeln('You can''t be serious!')
end.
```

Note that expressions such as

```
ch = 'y'
```

are perfectly legal Boolean expressions, and you can use them wherever any other Boolean expression would go. For example, to make sure the person using it actually types a "y" or an "n", you can add error checking to the program:

```
program character_lab;
{ experiments with characters }

var
 ch : char;

begin { character_lab }
 repeat
  writeln('Do you enjoy programming?');
  write('Please type y for yes, n for no:');
  read(ch);
  writeln;
  if (ch <> 'y') and (ch <> 'n') then
   writeln('I don''t understand...')
  else if ch = 'y' then
   writeln('Ah, I''m glad to hear it.')
  else { must be 'n' }
   writeln('You can''t be serious!')
 until (ch = 'y') or (ch = 'n')
end.
```

The other four relational operators (<, >, <=, >=) can be applied to characters as well, but before we try it, some explanation of how they work is in order.

The collection of all the different characters used by a given computer is called that computer's *character set.* It is helpful to think of characters as being really represented inside the computer's memory by small integer values. For example, on the Macintosh, the capital letter D is represented by the number 68. The character-as-number representation is often called a *collating sequence,* because it implies an ordering for the set of characters the computer uses. Different computers use different character sets, but the most common

is the one used on the Macintosh; it is known as the *ASCII character set.*

The Pascal relational operators compare characters based on the underlying ordering of the character set. For example, you can change the number-guessing game of Chapter 4 into a letter-guessing game. The program would go like this:

```
program guessing_game;
{ Guess a letter thought of by Macintosh }

var
 target, guess : char;
 success : Boolean;

begin { guessing_game }
 target := chr(random mod 26 + ord('a'));
 writeln('I am thinking of a letter between a and z');
 success := FALSE;
 repeat
  write('Your guess?:');
  read(guess);
  writeln;
  if guess < target then
   writeln('Sorry: ', guess : 1, ' is too low')
  else if guess > target then
   writeln('Sorry: ', guess : 1, ' is too high')
  else { got it ! }
   success := TRUE
 until success;
 writeln(guess : 1, ' is correct!')
end.
```

(The calculation of the "target" character in this program involves some features of Pascal we won't discuss until the next chapters.) All lowercase letters have a continuous sequence of values in the ASCII character set: the letter a comes before b, which comes before c, and so on. We will explore the ASCII set further in the next chapter.

One common mistake with characters involves confusing a *digit character* (such as '3') with the corresponding *integer value* (3). Characters and numbers are totally different beasts, and Pascal will reward any attempt to confuse them with bug boxes.

It is also important to remember that upper- and lowercase letters have distinct values and that Pascal doesn't acknowledge any special connection between them. For example, to test whether a character is an upper- or lowercase A, you must spell out both comparisons explicitly:

```
if (ch = 'A') or (ch = 'a') then
 ...
```

You may use a character as a loop control variable in a **for** loop. For example, this program will print out all the lowercase letters:

```
program character_lab;
{ experiments with characters }

var
 ch : char;

begin { character_lab }
 for ch := 'a' to 'z' do
  write(ch)
end.
```

How would you change this program to print the uppercase letters instead? Try it. Can you print the letters in reverse order? (Hint: remember **downto**?) On many computers, the "lowest" printable ASCII character is the space and the "highest" is the tilde (the ~ key in the upper-left corner of the Macintosh keyboard). Try replacing the lower and upper limits on the **for** loop with a space character and a tilde character, respectively. What happens?

You may also use a character expression as the controlling expression of a **case** statement. For example, here is a program that classifies input characters using **case**:

```
program classify;
{ classify an input character }

var
 ch : char;

begin { classify }
 write('Please type a character:');
```

```
read(ch);
writeln;
case ch of
 'a', 'e', 'i', 'o', 'u' :
  writeln('That''s a vowel.');
 'b', 'c', 'd', 'f', 'g', 'h', 'j', 'k', 'l', 'm', 'n', 'p', 'q', 'r', 's', 't', 'v', 'w',
 'x', 'z' :
  writeln('That''s a consonant');
 'y' :
  writeln('Sometimes y is a consonant, other times a vowel.');
 '0', '1', '2', '3', '4', '5', '6', '7', '8', '9' :
  writeln('That''s a digit.');
 otherwise
  writeln('I don''t know how to classify that . ')
end
end.
```

Try this program. Note that it recognizes consonants, vowels, and digits, but it considers everything else to be unclassifiable. How would you teach it to recognize math symbols (+, −, *, /), punctuation, and so on?

MACINTOSH PASCAL STRINGS

The character data type is an important addition to Pascal, but you may have doubts about its usefulness. After all, single characters don't often appear by themselves. In most applications, characters are grouped into words, words into lines, lines into paragraphs, and so on. Macintosh Pascal provides a data type that allows your program to treat a group of characters as a single variable: the *string* data type.

To declare string variables, simply use the reserved word **string** as the type:

```
var
 prompt, name : string;
```

You already know how to write string constants from our discussion in Chapter 1. Simply enclose the string characters in apostrophes. (The usual double-apostrophe rule for putting single apostrophes in the string also applies.) For example:

```
prompt := 'What"s your name?:'
```

String input and output really don't involve anything new. As you know, string constants may be displayed using write and writeln. (You have been doing this since our very first Pascal program.) You may also display string variables using write and writeln to specify field-width parameters, if you want. You may use readln to accept a string of characters from the keyboard and store those characters in a single string variable. This program demonstrates simple "echoing" of an input string:

```
program string_lab;
{ experiments with strings }

var
 prompt, name : string;

begin { string_lab }
 prompt := 'What"s your name?:';
 write(prompt);
 readln(name);
 writeln('Hi ', name, ', how"s it going?')
end.
```

Run this program and type your name when asked. As with character input, do not enclose the name in apostrophes. (What happens if you do?)

A string may contain from 0 to 255 characters. When you declare a string variable, enough space is allocated in memory to store 255 characters even though the number of characters actually stored in the string variable may be fewer than 255. The actual number of characters stored in the string is called the string's *length*, and the capacity of the string—the maximum number of characters it can hold—is the string's *size*. Don't confuse these two terms: a string's size is set when it is declared and remains constant throughout the program. A string variable's length, on the other hand, may vary while the program is executing, depending on the characters stored there.

When you just use the word **string** to declare a string variable (as we have been doing so far), the string's size defaults to 255. If you want to use a smaller size string, follow the word **string** with a size in square brackets, as shown here:

```
var
 prompt : string[18];
 name : string[80];
```

This says the string variable "prompt" has a size of 18, and the string "name" has a size of 80. Think of a string-size specification as a promise to Pascal that your program will never attempt to store more characters than specified into the string. (If you break such a promise, Pascal will respond with a run-time bug box.) Strings of different lengths must be specified in separate variable definitions.

You may also compare any two string values using the six Pascal relational operators. Like characters, the ordering is based on the underlying ASCII character set. Two strings are compared character-by-character starting at their first characters. If the first characters are the same, the second characters are compared, and so on. The comparison is based on the ASCII values of the first pair of unequal characters. If the two strings are of different lengths and are the same up to the end of the shorter string, the shorter string is considered to be "less than" the longer. Two strings are only considered to be equal if they are the same length and contain the same characters in the same order.

It is perhaps easiest to see how strings are compared by writing a short program:

```
program string_lab;
{ experiments with strings }

 var
  w1, w2 : string;

begin { string_lab }
 write('Enter a word:');
 readln(w1);
 write('Enter another word:');
 readln(w2);
 if w1 > w2 then
  writeln(w1, ' is greater than ', w2)
 else if w1 < w2 then
  writeln(w1, ' is less than ', w2)
 else { must be equal }
  writeln(w1, ' equals ', w2)
end.
```

Type in and run this program, entering your own test strings. Once you think you know how string comparison works, try to predict the results from entering these strings:

```
apple & banana
apple & Banana
apple & apple
apple & Apple
apple & apple pie
apple & ape

4 & 5
4 & 3
4 & 3121
```

Pascal's rules on type mixing are relaxed slightly to blur the distinction between the character and string types. You may compare a character value and a string value using the relational operators; the character is considered to be a "string of length 1" for purposes of comparison. A single-character constant ('g', for example) may be considered to be *either* a character constant or string constant, depending on the context. You may assign a character variable's value to a string variable; you may also assign a string variable's value to a character variable *if* the string variable contains exactly one character ("is of length 1").

You may not, however, use a string as a loop control variable in a **for** statement. Neither can you use a string expression to control a **case**.

It is possible to have a string variable that has zero length, that is, contains no characters. This special string value is called the *null string.* A null string is represented in a program by two consecutive apostrophes. For example, this assignment sets the string variable "name" to the null string:

```
name := ''
```

A common use for the null string is to signal the end of input from the keyboard. A null string is "typed" by just pressing the RETURN key without typing any other characters first. When it isn't known in advance how many lines of input are going to be entered in a program, the program may test each input string against the null string and exit when the test is satisfied. For example, the following program "echoes"

each input string until the null string is entered.

```
program echo;
{echo input until null string is typed }

var
  s: string;

begin { echo }
 repeat

   write('Enter a string (or return to stop):');
   readln(s);
   if s <> '' then
     writeln('You typed: ', s)
 until = s = ''
end.
```

A final note on portability: Standard Pascal offers no built-in string data type. In fact, a number of things we have discussed in this section are strictly illegal in Standard Pascal. So there is no guarantee that the Macintosh Pascal program you write today using strings will run tomorrow under another version of Pascal. (There is hope, however. Many versions of Pascal, including UCSD Pascal and Apple Pascal, provide a very similar string data type.)

If portability is your goal, you will have to shun Macintosh Pascal strings. However, you will find it is much harder (but not impossible) to write many programs without using the Macintosh Pascal string data type. In writing your own programs you will have to decide which is more important, program portability or programming convenience.

In the next chapter we will discuss a number of string-manipulation commands provided by Macintosh Pascal.

MACINTOSH PASCAL LONG INTEGERS

Another non-portable feature provided by Macintosh Pascal is the *long integer* data type. You can think of variables of the long integer type as higher-capacity integers; they can hold

larger values than can variables of the normal integer type we have already discussed. How much larger? To find out, try the following program:

```
program long_lab;
{ experiments with long integers }

begin { long_lab }
 writeln('MAXINT is ', MAXINT : 1);
 writeln('MAXLONGINT is ', MAXLONGINT : 1)
end.
```

MAXINT and MAXLONGINT are predefined constants available to any program you write; they give the maximum positive values of normal and long integer types:

```
MAXINT is 32767
MAXLONGINT is 2147483647
```

Both types of integer allow an equally large range of negative values. So, as we discussed in Chapter 2, normal integers are restricted to the range −32,767 up to 32,767. Long integers allow much bigger values, from −2,147,483,647 to 2,147,483,647.

To declare long integer variables in a Macintosh Pascal program, use longint as the type name:

```
var
 li, lj : longint;
```

Long integer values may be used just like normal integers. That is, all arithmetic and comparison operations you have been using on integers also work on long integers, in precisely the same way. Try the following program:

```
program long_lab;
{ experiments with long integers }

 var
  li, lj : longint;

begin { long_lab }
 li := 47931;
```

```
  lj := 35820;
  writeln('li is ', li : 1);
  writeln('lj is ', lj : 1);
  writeln('li + lj is ', li + lj : 1);
  writeln('li - lj is ', li - lj : 1);
  writeln('li * lj is ', li * lj : 1);
  writeln('li div lj is ', li div lj : 1);
  writeln('li mod lj is ', li mod lj : 1)
end.
```

You may also write long integer constants as hexadecimal numbers by preceding the constant with a dollar sign, for example:

```
li := $10fa23c
```

Everything you have learned about reading and writing normal integer values also applies to long integers. Long integer variables may be used for loop control in a **for** and long integer expressions can be used as **case** controllers.

When evaluating integer arithmetic expressions, Macintosh Pascal converts all normal integer values to long integer values before carrying out the arithmetic operations. This happens even when all variables involved are normal integers. For example, consider the simple assignment:

```
k := i + j
```

When this statement is executed, the values of the normal integer variables i and j are converted to long integers before they are added. The result is also a long integer; this is converted back into a normal integer and stored in the variable k.

This automatic conversion means that Pascal's type-mixing rules are effectively repealed when it comes to distinguishing between normal and long integers. You may mix the different types of integer together in expressions and assignments pretty much in whatever combinations you find convenient. You will cause a run-time error, however, if you attempt to store long values bigger than MAXINT in a normal integer variable.

One final caveat: long integers are non-portable. Standard Pascal only provides the normal integer data type. To minimize portability problems, it seems prudent to use long integers explicitly only when their greater capacity is needed.

MACINTOSH PASCAL REAL TYPES

If you were able to handle the concept of two different varieties of integer, you will have no trouble with the four different kinds of real numbers offered to you by Macintosh Pascal. These are the *real* type (already discussed in Chapter 2), the *double* type, the *extended* type, and the *computational* type. You will remember that values of the real data type discussed in Chapter 2 were limited both in their precision and range. The other three kinds of real numbers offer additional precision or range. These properties are summarized in Table 6-1. (Figures in Table 6-1 for precision and range are approximate only; for more precise numbers, use the methods developed in Chapter 2.)

As was the case for long and normal integers, the four real varieties are more alike than different. You declare variables of each type by using the appropriate type name in the variable declaration section:

```
var
 x : real;
 y : double;
 z : extended;
 w : computational;
```

All four real types participate in all the mathematical operations you have already used for the normal real data type. All real-type variables and constants used in expressions are converted to extended real values before the mathematical operations are performed, just as integer values are converted to long integer values.

Table 6-1.

Range and Precision of Real Types

Real Type Name	Smallest Positive Value	Largest Positive Value	Precision (Decimal Digits)
real	1.5×10^{-15}	3.4×10^{38}	7-8
double	5.0×10^{-324}	1.7×10^{308}	15-16
extended	1.9×10^{-4951}	1.1×10^{4932}	19-20
computational	1	9.2×10^{18}	(exact)

As was the case for the two integer types, the four real types are essentially exempt from Pascal's usual type-mixing rules. You may legally assign a value of any real type to a variable of any real type. The only restriction is that you may not assign a value to a variable that doesn't fit into the range of that variable's type; a run-time error results if you try.

The computational type differs slightly from the other real types in that it is defined to represent only *integral values:* numbers without a fractional part. It is particularly useful when large numbers must be represented precisely without the rounding errors inherent in normal real arithmetic. The typical use for such preciseness is in calculations involving monetary amounts; such amounts can be represented as whole numbers by representing numbers of pennies. (For example, the amount $19,342,542.45 would become the computational value 1934254245.) When used this way, the computational type would allow monetary amounts up to approximately 92 quadrillion dollars ($\$9.2 \times 10^{16}$) to be represented exactly (very useful if you are using the computational type to keep track of your checking account).

You may read and write real values of all types just as you have always done for normal real values. For example, the following program:

```
program comp_demo;
{ show computational output }

 var
  price, tendered, change : computational;

begin { comp_demo }
 price := 1999;
 tendered := 2000;
 change := tendered - price;
 writeln('Price:' : 12, price / 100 : 10 : 2);
 writeln('Tendered:' : 12, tendered / 100 : 10 : 2);
 writeln('Your change:' : 12, change / 100 : 10 : 2);
 writeln('Thank you for shopping Programs-R-Us.')
end.
```

would produce this output:

```
      Price:      19.99
   Tendered:      20.00
```

```
Your Change:       0.01
Thank You for shopping Programs-R-Us
```

The Macintosh Pascal documentation describes a special feature used for output of a single computational variable: the digits-after-the-decimal parameter in the write or writeln statements may be used to insert a false decimal point into the output value. Despite the documentation, this feature is not available in Release 1.0 of Macintosh Pascal. If you have a later version than 1.0, it may be present. An easy way to find out is to omit the division by 100 in the writeln statements in the comp_demo program; if it works as before, the feature has been added.

Programs can be easily modified to take advantage of the additional range and precision of the new real types. For example, let's modify the Newton program from Chapter 4 to use extended real values instead of normal reals. All that's needed is to substitute the word "extended" for "real" everywhere and to modify the writeln statement to print out the result with its newly acquired precision. Retrieve Newton and alter it as follows:

```
program Newton;
{ Calculate square roots by Newton's method }

 var
  x, old_guess, new_guess : extended;

begin { Newton }
 write('Enter a number:');
 readln(x);
 if x > 0.0 then
  begin
   new_guess := x / 2;
   repeat
    old_guess := new_guess;
    new_guess := (old_guess + x / old_guess) / 2
   until new_guess = old_guess;
   writeln('The square root of ', x : 27, ' is ', new_guess : 27)
  end
 else if x = 0.0 then
  writeln('The square root of ', x : 27, ' is ', x : 27)
 else
```

```
    writeln('Sorry, this program only works for non-negative
    numbers.')
end.
```

Now use this program to calculate the square root of 2; the result should look like this:

```
The square root of
2.00000000000000000e+0
is 1.41421356237309504e+0
```

This result is correct out to the final decimal place printed. Can this program calculate the square root of 10^{4932} (enter this as 1e4932)? Try it, but be patient. (Do you see why you will need to be patient? Use the Observe window to watch the "progress" toward a solution.)

Standard Pascal only provides a single real type, the normal type we discussed in Chapter 2. Unless your program requires the extended precision or range of the other real types, it is a good idea to restrict yourself to using the normal real type for portability's sake.

A DIGRESSION ON READ AND READLN

In a number of our programs we have used the read and readln commands to get data typed at the keyboard. We've seen that read and readln can be used for input values of every type: integer, real, Boolean, character, string, and long integer. Read and readln have slightly more complex behavior than we have discussed so far, however; you deserve a more complete description.

You may use a single read or readln statement to read in the values for a number of different variables, not just one. The rule is as follows: the read statement that specifies a list of variables to be read,

```
read(v1, v2, ..., vn)
```

is equivalent to a series of read statements contained in a **begin...end** compound statement:

```
begin
 read(v1);
 read(v2);
 ...
 read(vn)
end
```

As far as Pascal is concerned, the input data typed at the keyboard is a sequence of characters arranged into one or more lines. Each individual read starts off where the previous one stopped.

A readln command may appear all by itself,

```
readln
```

with no variable names (just as writeln can). When an "all-by-itself" readln is executed, Pascal skips to the next line of input. It will wait for you to type RETURN (if you haven't already done so). Any subsequent reading will be from the next line.

Readln can also be provided with a list of variables to read; the evaluation rule is similar to that of read. The readln statement

```
readln(v1, v2, ..., vn)
```

is defined to be equivalent to the compound statement:

```
begin
 read(v1);
 read(v2);
 ...
 read(vn);
 readln
end
```

So any single complex read or readln statement can be broken down into a series of individual reads. Each individual read will behave differently, however, based on the type of variable involved. You'll find these variations in the rules of read operations:

- *Reading characters.* This is the simplest case. When a character value is to be read, Pascal just accepts the

next keypress as the value. This means, of course, that you can't use the BACKSPACE key to correct an erroneous character; Pascal reads the first character you type and goes with it. If you press the RETURN key, it is read as a space character.

- *Reading strings.* Pascal will read all characters up to, but not including, the next RETURN into the string. BACKSPACE works to correct mistakes. A run-time error occurs if the number of characters typed exceeds the declared size of the string variable.
- *Reading numbers.* Pascal expects numbers to begin either with a sign or a digit character, so it will patiently wait until you type one of those characters; any other characters you type will be ignored. (Spaces and RETURN characters will be shown in the Text window; anything else will cause the speaker to beep.)

  Once Pascal recognizes a sign or digit character, it begins treating following digits as part of the number. BACKSPACE works to correct mistakes. It continues reading until it sees some character that can't possibly be part of the number, which ends the read operation.

  These same rules apply to reading both normal and long integers and all four varieties of real (although the allowable syntax for reals and integers differs). A run-time error will occur if the number read is outside the legal range for the variable type.
- *Reading Booleans.* Pascal reads the next identifier, using the Pascal identifier syntax rules discussed in Chapter 1. Leading characters that can't legally begin an identifier (non-letters) are ignored. The read operation starts when a letter is seen and terminates when a character is read that can't be part of a legal identifer (that is, a character that is neither a letter nor a digit). BACKSPACE may be used to correct typing errors. A run-time error occurs if the identifier read is not TRUE or FALSE. (Case is ignored; "True" and "False" are acceptable inputs.) Portability note: reading Boolean values is not possible in Standard Pascal.

Editing your input using the BACKSPACE key, as noted, never works for character input. Neither can it be used *after* an individual value has been read, even if that value is on the same line as the cursor.

These input rules are not easy to remember, and they have subtle implications you might not appreciate. Test the rules, if you like, using the following program:

```
program read_lab;
{ experiments with read }

 var
  s : string;
  c : char;
  i : integer;
  r : real;
  b : Boolean;
  l : longint;

begin { read_lab }
 readln(l, b, i, c, s, r);
 writeln('I read:');
 writeln('l = ', l);
 writeln('b = ', b);
 writeln('i = ', i);
 writeln('c = "', c, '"');
 writeln('s = "', s, '"');
 writeln('r = ', r);
end.
```

Enter this program and run it. Here are three possible sets of input you can use to try it out. In each case, try to predict the outcome before you see it:

```
444444FALSE 23
Apple Pie 23.62
98.6
```

```
   98632
TrUe!6!Orange
98632 21.341
```

```
234,TRue,432,Apple,32
45e-12,5,Thomas Jefferson
```

For further experiments, you might try causing the run-time errors mentioned in the list of input rules. Also try to observe when BACKSPACE editing is legal and when it's not. Finally, you might try different orderings of the variables in the readln statement and see how the rules for input are affected.

In writing programs of your own, two of the goals you should strive for are reliability and ease of use. Any program that crashes at the slightest provocation or has a myriad of complex rules that anyone using it must remember is unacceptable. Most of your battles for program reliability and ease of use will be fought in the part of your program that accepts input from the user. Here are a few suggestions you can use as guidelines:

- All input should be requested by using a clear and concise prompt, explaining precisely what sort of information is needed.
- Don't ask for more than one piece of input information at a time.
- Check as completely as possible for mistaken input; if your program expects "a number between 1 and 10," for example, put in an **if** statement to ensure that the number typed in is actually in that range. This also applies to input from yes-or-no questions.

INTRODUCTION TO LIBRARY FUNCTIONS

7

> Library functions...are one way to reduce the apparent complexity of a program; they help to keep program size manageable, and they let you build on the work of others, instead of starting from scratch each time.
>
> *BRIAN KERNIGHAN* and *P.J. PLAUGER*
> *The Elements of Programming Style*
> (McGraw-Hill, 1978)

You may remember Newton, the square root calculator program we wrote in Chapter 2 and modifed in subsequent chapters. That program did a reasonably good job of calculating square roots quickly, using about five lines of Pascal code, excluding input, output, and error-checking statements. But what if you were to write a program that required the calculation of a square root at many different points? Based on our discussion so far, you would need to rewrite those same five lines over and over each time a square root calculation was needed, certainly a tedious and error-prone task.

Fortunately, Pascal provides an easier way. Common, repetitive calculations (or other tasks) may be carried out in *subprograms*, instead of in the main program itself. A subprogram may be thought of as a little program used to perform some chore that helps achieve the overall goal of the program.

Subprograms are vital to most Pascal programs; often, a large program will consist mainly of subprograms. Macintosh Pascal provides a number of subprograms you can use in

writing your own programs, which we'll call *library* subprograms. Library subprograms perform common, useful, and often complex tasks.

Pascal offers two slightly different types of subprograms: *procedures* and *functions*. As we'll see, procedures and functions primarily differ in how they are used. Although there is no concrete rule, functions are often used to calculate values, while procedures are usually said to perform actions. In this chapter we will examine some of Macintosh Pascal's library functions; the next chapter introduces some of the library procedures. Both types of library subprograms make development of more sophisticated programs much easier.

STANDARD FUNCTIONS

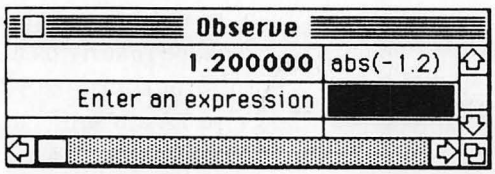
The simplest type of Pascal subprograms are the so-called *standard functions*. Standard Pascal provides for 17 standard functions; we will show 15 of them in this section. The standard functions are also known as *required functions;* they are required to be present in every version of Pascal.

To start, let's concentrate on a single standard function: the *absolute value* function. You may remember from mathematics that the absolute value of a number is simply the number with its sign removed—the absolute value of −1.2 is 1.2, for example. The absolute value of a positive number is the same as the number itself—the absolute value of 34 is just 34.

Run Macintosh Pascal, then open the Observe window by choosing Observe from the Windows menu. Type the following line into the right side of the Observe window and press the ENTER key:

```
abs(-1.2)
```

You should see the following result after a bit of disk whirring:



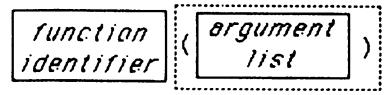


Figure 7-1.

Function call syntax

Can you see what's going on? Remember, the Observe window *evaluates expressions;* as far as Pascal is concerned, "abs(−1.2)" is an expression that can be evaluated to the specific result 1.200000. To see that "abs" behaves like the mathematical absolute value, type

```
abs(1.2)
```

into the next box of the Observe window. Then press ENTER as usual. Is the result as you expected?

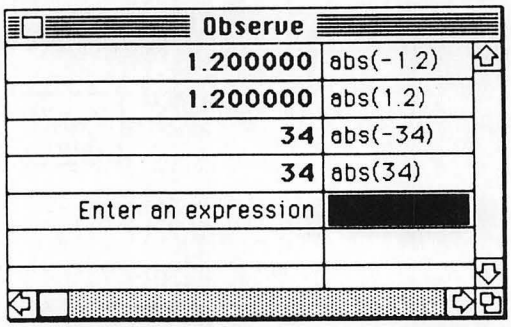
A small amount of terminology needs introduction here. The string "abs(−1.2)" is an example of a *function call.* A function call consists of two parts: a *function identifier* (abs, in this case) and an optional *argument list* (which consists of the single argument, −1.2, in this case). The argument list, if present, is enclosed in parentheses. Figure 7-1 shows the syntax of function calls.

More formal texts on Pascal refer to arguments as *actual parameters.* You will also often see the value of a function call referred to as the function's *result* or *return value.*

The important thing to remember about function calls is that they represent values. You already know that all values in Pascal have types (integer, real, character, and so forth); function call values are no exception. The abs function call, as you have seen, represents a real value when it is supplied with a real argument. If supplied an integer argument, it stands for an integer value. Try typing the following expressions into the Observe window to verify this:

```
abs(-34)
abs(34)
```

The results should appear as follows. (Note we have expanded the window to allow viewing all values simultaneously.)



Since function calls represent values, they may legally appear anywhere in an expression where variable identifiers or constants of the same type could appear. So function calls may be combined with other operations to form arbitrarily complex expressions. For example, try out the following expressions in the Observe window. (As always, try to predict the results first.)

```
9 * abs(-0.8)
abs(-3) - abs(-2)
```

Finally, you are not restricted to specifying single numbers as the arguments to a function. In general, whenever a function call expects an argument *value* of a certain type, you may substitute an *expression* of the same type. The value of the expression is the actual argument used by the function. To see how this works, try the following expressions in the Observe window:

```
abs(4.6 - 5.8)
abs(2 + 16 div 5 - 3)
```

As another example of a standard function, let's look at the *square root* function; its function identifier is the word "sqrt." Type the following expressions into the Observe window:

```
sqrt(2)
sqrt(2.0)
sqrt(64)
```

The results should appear as follows:

1.414214	sqrt(2)
1.414214	sqrt(2.0)
8.000000	sqrt(64)

Note that the square root function returns a real result even when its argument is an integer value (even a perfect square). As you might expect, it is illegal to take the square root of a negative number. (What happens if you try it?)

To show that functions need not always have numeric results, try out the function identifier *odd*. Type the following expressions into the Observe window:

```
odd(53)
odd(96)
odd(436210 div 2)
```

The results are these:

True	odd(53)
False	odd(96)
True	odd(436210 **div** 2)

The odd function accepts an integer value (either normal or long) as its argument and returns a Boolean value: TRUE if the argument is odd, FALSE if even.

Table 7-1 is a list of the standard functions most likely to be used in mathematical calculations. Note that nearly all of these functions accept either real- or integer-type arguments. Functions noted as accepting integer argument values will accept either normal or long values; functions marked as accepting real values will accept any of the four types of real values. (Actually, as noted in Chapter 6, *all* integer expressions, including arguments and function calls, are automatically converted to long integer values. All real arguments and function calls are likewise converted to extended real values.)

It's easy to use these functions in actual programs. For example, let's write a program to solve quadratic equations of the form

$$ax^2 + bx + c = 0$$

Depending on the actual values of a, b, and c in the equation, there are two solutions given by the formula

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a}$$

The squaring and square root operations in this formula can be performed using the library functions sqr and sqrt. A

Table 7-1.

Standard Arithmetic Functions	
abs(x)	
argument:	x-real or integer value
result type:	same as argument
description:	absolute value, |x|
arctan(x)	
argument:	x-real or integer value
result type:	extended real
description:	trigonometric arctangent, $\tan^{-1}(x)$
cos(x)	
argument:	x-real or integer value
result type:	extended real
description:	trigonometric cosine, cos(x) (x in radians)
exp(x)	
argument:	x-real or integer value
result type:	extended real
description:	exponential, e^x
ln(x)	
argument:	x-real or integer value
result type:	extended real
description:	natural logarithm, $\log_e(x)$ $(x > 0)$
odd(i)	
argument:	i-integer value
result type:	Boolean
description:	TRUE if argument is odd, FALSE if even
sin(x)	
argument:	x-real or integer value
result type:	extended real
description:	trigonometric sine, sin(x) (x in radians)
sqr(x)	
argument:	x-real or integer value
result type:	same as argument
description:	square, x^2
sqrt(x)	
argument:	x-real or integer value
result type:	extended real
description:	square root, $\sqrt{x}$ $(x > 0)$

simple program to accept input values for a, b, and c and then to calculate and print the solutions to the corresponding quadratic equation might go like this:

```
program quadsolver;
{ solve quadratic equations }

var
 a, b, c : real;
 discrim : real;
 root1, root2 : real;

begin { quadsolver }
 writeln('This program solves quadratic equations.');
 write('Enter value for a: ');
 readln(a);
 write('Enter b:');
 readln(b);
 write('Enter c:');
 readln(c);
 discrim := sqr(b) - 4.0 * a * c;
 root1 := (-b + sqrt(discrim)) / 2.0 / a;
 root2 := (-b - sqrt(discrim)) / 2.0 / a;
 writeln('The roots are:', root1 : 16, ' and ', root2 : 16)
end.
```

Type in and run this program. Use it to solve the following equation:

$$x^2 - x - 1 = 0$$

For this equation, the values of a, b, and c are 1, −1, and −1, respectively. Compare your results with those obtained by the program we wrote to solve this equation in Chapter 5.

Notice that this program will crash if either the value of a is 0 or the argument to the square root function is less than 0. As an exercise, modify it to check for and avoid these situations.

Standard Pascal also provides two functions whose purpose is to perform real-to-integer conversion; they are called *round* and *trunc*. Loosely speaking once more, the trunc function "chops off" any fractional part of a real number, resulting in an integer value. The round function, on the other

Table 7-2.

Standard Transfer Functions	
round(x)	
argument:	x-real value
result type:	long integer
description:	nearest whole number
trunc(x)	
argument:	x-real value
result type:	long integer
description:	nearest whole number between 0 and x. (Removes fractional part)

hand, "rounds off" the real value to the nearest integer value. Since they perform type conversion, these two functions are called *transfer functions.* Their definitions are found in Table 7-2.

Try out the following expressions in the Observe window to see how the two transfer functions compare in their workings:

```
trunc(2.9)
round(2.9)
trunc(3.14)
round(3.14)
trunc(-3.14)
round(-3.14)
trunc(-2.9)
round(-2.9)
round(sqrt(6.25))
```

(Note there is nothing wrong with using a function call in the argument to another function.) The results should appear as follows:

2	trunc(2.9)
3	round(2.9)
3	trunc(3.14)
3	round(3.14)
-3	trunc(-3.14)
-3	round(-3.14)
-2	trunc(-2.9)
-3	round(-2.9)
3	round(sqrt(6.25))

A third class of standard functions is known as the *ordinal functions*, so called because they apply only to ordinal types. We have seen four ordinal types so far: Booleans, integers, long integers, and characters. None of the four real types (not even computational) nor the **string** type is an ordinal type.

Any value of an ordinal type is defined to have an ordinality different from any other value of the type. The ordinality of an integer is just its value. The ordinality of a character is its representation in the computer's character set; on the Macintosh, the ordinality of a character is its ASCII value. Finally, Boolean ordinality is defined to be 0 for FALSE, 1 for TRUE.

Not surprisingly, Pascal provides a library function that calculates the ordinality of any value of an ordinal type; it is called *ord*. To see how it works, type the following expressions into the Observe window:

```
ord('a')
ord('A')
ord('3')
ord(3)
ord(FALSE)
ord(TRUE)
ord(-345678)
```

The results should appear as follows:

97	ord('a')
65	ord('A')
51	ord('3')
3	ord(3)
0	ord(FALSE)
1	ord(TRUE)
-345678	ord(-345678)

You have just seen how you can use ord to discover the ASCII value of any character. Pascal also provides the *chr* function that does just the opposite; it accepts an integer representing a character and returns that character as a result. The ASCII character set on the Macintosh runs from 0 to 255, so it is an error to pass an argument outside this range

to the chr function. Try the following expressions in the Observe window:

```
chr(97)
chr(65)
chr(192)
chr(ord('*'))
ord(chr(45))
```

You should obtain the following results:

a	chr(97)
A	chr(65)
¿	chr(192)
*	chr(ord('*'))
45	ord(chr(45))

The final two ordinal functions are called *pred* and *succ*. They accept any ordinal value as an argument and return the value before and after the value, respectively. To see how they work, try the following expressions in the Observe window:

```
pred(22)
succ(22)
pred('j')
succ('j')
pred(TRUE)
pred(succ('B'))
pred(pred(pred(pred('z'))))
```

You should observe the following results:

21	pred(22)
23	succ(22)
i	pred('j')
k	succ('j')
False	pred(TRUE)
B	pred(succ('B'))
v	pred(pred(pred(pred('z'))))

The pred and succ functions are also inverse functions. It is an error to try to find the predecessor of the first value of a

given ordinal type (or the successor of the last value). For example, trying to evaluate "pred(FALSE)" will give you a bug box.

The chr function can be used to produce characters that unfortunately can't be typed directly into a Macintosh Pascal program. Let's write a small program to write out all possible characters provided in the Macintosh's character set. A first attempt might go as follows:

```
program ASCII;
{ output ASCII character set }

 var
  i : integer;

begin { ASCII }
 for i := 0 to 255 do
  write(chr(i))
end.
```

Type in this program and run it. (The hollow boxes you see mixed in with the other characters are the Macintosh's way of signaling that those characters aren't available in the current font.) You'll notice the program isn't too satisfactory because the characters are printed out all jammed together. Let's make a couple of modifications to skip to a new line after a line of 16 characters has been printed and to print each character in a field width of 2 so that there will be at least one space between each character:

```
program ASCII;
{ output ASCII character set }

 var
  i : integer;

begin { ASCII }
 for i := 0 to 255 do
  begin
   write(chr(i) : 2);
   if i mod 16 = 15 then
    writeln
  end
end.
```

Try these modifications. (You may have to adjust the size of the Text window to see all the output.) Do you see why the **mod** calculation skips to a new line after each line of 16 characters? You'll also notice that the first two rows aren't nicely aligned as are the following rows; how can you identify the characters causing the problems?

As a final exercise, try the following mysterious program:

```
program mystery;
{ what does this do? }

begin { mystery }
 writeln('The Macintosh', chr(170), ' is in use.');
 writeln('It''s about 3', chr(161), ' below zero!');
 write('Plus ', chr(141), 'a change, c', chr(213));
 writeln('est la m', chr(144), 'me chose.');
 writeln(chr(185), ' ', chr(197), 4.0 * arctan(1.0) : 27)
end.
```

The ordinal functions are summarized in Table 7-3.

Table 7-3.

Standard Ordinal Functions

chr(i)	
argument:	i-integer value (0-255)
result type:	character
description:	character whose ordinality is i
ord(c)	
argument:	c-value of any ordinal type
result type:	long integer
description:	ordinality of c
pred(c)	
argument:	c-value of any ordinal type
result type:	same as argument
description:	predecessor of c
succ(c)	
argument:	c-value of any ordinal type
result type:	same as argument
description:	successor of c

MACINTOSH PASCAL STRING MANIPULATION FUNCTIONS

You now know how functions are used to represent values of different types in expressions. Just as Standard Pascal provides a number of useful functions that operate on standard types, Macintosh Pascal provides its own functions that allow manipulation of its own data types. In this section we'll concentrate on Macintosh Pascal's string manipulation functions.

Macintosh Pascal offers functions to

- Determine the length of a string
- Concatenate strings together
- Insert characters into strings
- Delete characters from strings
- Search a string for an occurrence of a smaller string
- Extract substrings from strings
- Convert values of all printable types into strings.

The string manipulation functions are summarized in Table 7-4.

Once more, it's easiest to try a few examples of the string manipulation functions in the Observe window if you are uncertain about their use. Try typing in the following examples:

```
length('Hello!')
concat('con', 'cat', 'ena', 'tio', 'n')
pos('e', 'applesauce')
include('niver', 'use', 2)
omit('usage', 3, 2)
copy('usage', 2, 3)
```

Your results should look something like this:

6	length('Hello!')
concatenation	concat('con', 'cat', 'ena', 'tio', 'n')
5	pos('e', 'applesauce')
universe	include('niver', 'use', 2)
use	omit('usage', 3, 2)
sag	copy('usage', 2, 3)

Table 7-4.

Macintosh Pascal String Manipulation Functions

concat(s1, s2, ..., sn)
- arguments: s1, s2, ..., sn-**string** values (one or more arguments)
- result type: **string**
- description: concatenation of the arguments

copy(s, i, n)
- arguments: s-**string** value; i, n-integer values
- result type: **string**
- description: substring of n characters starting at character #i of s (null string if n < 0)

include(s1, s2, i)
- arguments: s1, s2-**string** values; i-integer value
- result type: **string**
- description: s2 with the string s1 inserted at character #i

length(s)
- argument: s-**string** value
- result type: long integer
- description: length of s

omit(s, i, n)
- arguments: s-**string** value; i, n-integer values
- result type: **string**
- description: s with n characters deleted starting at character #i

pos(s1, s2)
- arguments: s1, s2-**string** values
- result type: long integer
- description: position of substring within s2 matching s1 (0 if no substring of s2 matches s1)

stringof(w1, w2, ..., wn)
- arguments: w1, w2, ..., wn—any write arguments (one or more arguments; each may have formatting parameters
- result type: **string**
- description: string containing written representation of arguments

Examine these results carefully to make sure you know why each function behaves as it does; try more examples in the Observe window if you're doubtful.

The *stringof* function described in Table 7-4 does not work in the Observe window because of a bug in Macintosh Pascal version 1.0. (If you have a later version, this bug may have

been fixed.) The stringof function accepts one or more arguments; the form of each argument to stringof follows the same rules you already know for arguments to the write and writeln commands. (In other words, every argument list you have used for write or writeln could also be used as an argument list for stringof.)

As you know, write and writeln convert their arguments into printable form—into characters—and output this printed representation to the Text window. Alternatively, the stringof function converts its arguments into printable form and returns this representation as a string value corresponding to that printed form. Unlike write and writeln, no output is performed. The stringof function may be useful when you need to print a value in a format not available from write or writeln. It will do the hard task of converting the value to character form, and you can use the other string manipulation functions to twiddle the resulting string into the format you want.

The string manipulation functions are, of course, useful for pulling apart strings and putting them back together in different ways. We will describe two simple programs here as examples and give you suggestions for programs to write on your own.

First, let's write a program to perform a simple task: read in an input string and print it back out with all the lowercase letters changed to the equivalent uppercase letters. As usual, a good place to begin is with a pseudo-code outline of the strategy. Especially in the early stages of program design, pseudo-code doesn't have to be particularly sophisticated; it can, and perhaps should, lean toward simplicity. For example, we could write the case-converter pseudo-code this way as a crude first step:

```
get string
covert all lowercase characters in string to uppercase
print converted string
```

Don't scoff at the simplicity of this strategy; the important thing is that the pseudo-code is obviously correct. We can be sure that any program we write that correctly implements this strategy will work.

Also note that we have taken our single large problem and split it into three smaller problems; presumably, each problem can then be solved more or less independently from the others. In this case, the first and third steps in the pseudo-

code are easy to implement in Macintosh Pascal: getting the string and printing the converted string can be done with the readln and writeln routines. So the second step, converting the string, is the main problem to solve. But it is at least a smaller, simpler problem to solve than the original.

In trying to get a handle on the solution, you might note that the lower- to uppercase conversion must be done on a character-by-character basis: the program has to examine each character in the string, test it to see if it is a lowercase letter, and, if it is, convert it to uppercase. So a refined pseudo-code might appear as follows:

```
get string
look at each character in the string:
    if character is a lowercase letter
        replace with the corresponding uppercase letter
print converted string
```

Now how will we look at each character in the string? Looking through Table 7-4 for ideas, we notice that the copy routine can be used to extract substrings from a string. Can copy be used to extract one character from the string? Of course. If the argument n has the value one, then the assignment statement

```
ch := copy(s, i, 1)
```

extracts the single character at position i of the string s into the character ch. Looking at *each* character, then, involves letting the variable i assume values from 1 (to access the first character in the string) up to the number of characters in the string (to access the last character in the string). This can be done most easily with a **for** loop, remembering that the length function gives the number of characters in a string:

```
for i := 1 to length(s) do
    ...
```

How about testing whether the character is a lowercase letter? Remember that the character values a through z are a contiguous sequence of values in the ASCII character set; that is, there are only lowercase letters between a and z. So the test for lowercase can be simply

```
if (ch >= 'a') and (ch <= 'z') then
  ...
```

The remaining problem in the pseudo-code is how to replace a lowercase letter with the equivalent uppercase letter. Looking to the string functions in Table 7-4, we fail to find a replace function, but there is an *omit* function that can be used to delete characters from a string and an *include* function to put characters into a string. Armed with these two operations you can perform a replacement. Let's rewrite the pseudo-code one more time:

```
get string
for each character in the string:
    extract character from string
    if character is a lowercase letter:
        delete character from string
        convert character to uppercase
        insert converted character into string
          (at the same place)
    end if
end for
print converted string
```

The final problem is how to go about obtaining the uppercase letter corresponding to the lowercase letter extracted from the string. You may have noticed in the previous section that the ASCII value of the character A is 65 and the ASCII value of a is 97. Since a to z and A to Z are contiguous ASCII values, it's not hard to see that the ASCII value of a lowercase letter is always 32 less than the ASCII value of the corresponding uppercase letter. Let's work through the necessary calculation a step at a time. If the character variable ch holds a lowercase letter, then

```
ord(ch)
```

is the ASCII value of the letter and

```
ord(ch) - 32
```

is the ASCII value of the corresponding uppercase letter. Its value is represented by

```
chr(ord(ch) - 32)
```

Another way to accomplish the same thing is

```
chr(ord(ch) - ord('a') + ord('A'))
```

Can you see why this expression is equivalent to the previous expression?

Putting all this together into a program is a straightforward translation of the now quite detailed pseudo-code:

```
program lower_to_upper;
{ convert lowercase letters in string to uppercase }

 var
  s : string;
  ch : char;
  i : integer;

begin { lower_to_upper }
 write('Enter a string:');
 readln(s);
 for i := 1 to length(s) do
  begin
   ch := copy(s, i, 1);
   if (ch >= 'a') and (ch <= 'z') then
    begin
     s := omit(s, i, 1);
     ch := chr(ord(ch) - ord('a') + ord('A'));
     s := include(ch, s, i)
    end
  end;
 writeln(s)
end.
```

Compare this program with the last version of the pseudo-code. Can you see how each part of the program arose from the corresponding pseudo-code? Type in and test the program; does it work on all possible input strings?

As our second example, let's write a program that accepts an arbitrary number of input lines and prints out the number of words in the input and the average word length. (As a practical application, this could become the core of a text-analysis program that measures the readability of a docu-

ment.) Once more, we start with pseudo-code that is nearly obvious:

```
while there's another input line
    read input line
    process input line
end while
print results
```

How will we determine when there are no more input lines? One simple scheme, which we'll use, is to let an empty input line signal the end of the input. We can determine whether a line is empty or not by comparing the line with the null string, or, more simply, testing to see if its length is 0. The pseudo-code can then be rewritten slightly:

```
read first input line
while input line is not empty
    process input line
    read next input line
end while
print results
```

Now the tough problem to solve is the line-processing strategy. Useful questions to ask yourself at this point in the design are: How would I do this problem in real life, without a computer? Could I describe my method in step-by-step fashion, simply enough so a computer could perform each step? Does the language I am using provide any help in the form of, say, library functions? Does my strategy account for all possible situations? Am I making any possibly incorrect assumptions about the form of the input? Try, on your own, to come up with a refined pseudo-code for the line-processing problem before you read on.

There are actually a number of possible methods you might consider in solving this problem. Our strategy—not necessarily the best one—will be to repeatedly look for the first word in the input line. As we find each word, we can extract it from the input line, measure its length, then delete it from the input line. Eventually we will wind up with no words left in the input line; that will signal that we are done processing the line.

Putting this refined description into pseudo-code might go as follows.

```
read first input line
while input line is not empty
    repeat
        extract next word from line
        measure its length
        delete word from line
    until no words left in line
    read next input line
endwhile
print results
```

So far, so good. A useful next step might be to get a little more specific on the actual arithmetic the program will have to do. In order to calculate an average, we will need to keep track of both the number of words and the sum of the word lengths seen. We'll need two variables to keep track of these two quantities. Each variable should be set to zero initially, and each time a word is found the values of each variable must be updated. The pseudo-code now looks like this:

```
set word count to 0
set sum of word lengths to 0
read first input line
while input line is not empty
    repeat
        extract next word from line
        delete word from line
        add length of word to sum of word lengths
        add one to word count
    until no words left in line
    read next input line
endwhile
print word count
print average word length
    ( = sum of word lengths/word count)
```

Let's now consider how to go about extracting the next word from the line. Before we start, however, we'll need to specify more precisely what we mean by a word. For now, let's adopt a somewhat simplistic, but workable, definition: a *word* is any sequence of characters excluding blanks. That is, blanks can be considered as word separators.

Now is also a good time to consider the possible "strange" input lines our program might have to deal with. It's easy to

see the program will have to properly handle lines of different lengths, each possibly with different numbers of words. But might a line contain zero words? Yes, if it has only blanks in it. Should our program assume there will always be exactly one blank between words? Probably not; instead, it should assume words might be separated by any number of blanks. Might the input line be entered with leading or trailing blanks? It might happen, in which case our program should deal with them intelligently.

All these considerations guide the next step in refining the pseudo-code:

```
set word count to 0
set sum of word lengths to 0
read first input line
while input line is not empty
    repeat
        delete leading blanks from line (if any)
        if there are more characters in the line
            search for next blank (or end of line)
            extract word: all characters from line
                  up to next blank (or end of line)
            delete word from line
            add word length to sum of word lengths
            add one to word count
        endif
    until no words left in line
    read next input line
endwhile
print word count
print average word length
    ( = sum of word lengths/word count)
```

At last we have reached the stage where the pseudo-code can be translated nearly line-by-line into Pascal. You may want to try to do this yourself before you examine the following program:

```
program wordcount;
{ count words in input text }

 var
  line, word : string;
  nextblank, wordcount, sumlength : integer;
```

```
begin { wordcount }
 write('This program counts the number of words in an ');
 write('arbitrary number of input lines and computes ');
 writeln('the average word length.');
 writeln;
 write('Please enter the input text. Press the Return ');
 write('key after each line. Enter an empty line when ');
 writeln('done.');
 writeln;
 sumlength := 0;
 wordcount := 0;
 readln(line);
 while length(line) > 0 do
  begin
   repeat
    while pos(' ', line) = 1 do { delete leading blanks }
     line := omit(line, 1, 1);
    if length(line) > 0 then
     begin
      nextblank := pos(' ', line);
      if nextblank = 0 then { no blanks left in the line }
       nextblank := length(line) + 1;
      word := copy(line, 1, nextblank - 1);
      line := omit(line, 1, nextblank - 1);
      wordcount := wordcount + 1;
      sumlength := sumlength + length(word)
     end
   until length(line) <= 0;
   readln(line)
  end;
 writeln('Number of words: ', wordcount : 1);
 write('Average word length: ');
 writeln(sumlength / wordcount : 1 : 1)
end.
```

If you wrote your own version of this program, compare it with this one. If they differ, can you say which is better than the other? It may well be that your version is an improvement.

Enter the program—either your own version or this one—into the Macintosh and test it using different sets of input. Make sure to enter some simple cases (only one or two words) so you can verify that the calculation is being done correctly. In addition, try all the strange input cases discussed above:

leading and trailing blanks, lines containing only blanks, and lines with more than one blank between words.

You may have already found the logic error in our version of the program; if not, try to find and correct it now. (Hint: can the program ever get a division-by-zero error?)

You may also want to consider correcting a rather poor feature of the program resulting from our too simple definition of what a word was: punctuation characters adjacent to a word are considered to be part of the word. Modify the program to delete punctuation before a word's length is measured. (How will you define punctuation precisely? Is it enough to consider leading and trailing punctuation? What is the length of, say, the word "isn't"?)

The purpose of going into such detail over the program design process was to emphasize two facts: first, writing programs is not an automatic process; it demands near fanatic devotion to detail and an ability to break a seemingly difficult problem into smaller and smaller subproblems. Worse, things will seldom go as smoothly as shown here. More often than not, the programming process goes down blind alleys; the usual solution is to tear up the non-working parts of the design and start over.

The second fact—the good news—is that there really isn't anything more than that involved. Acquiring programming skill is mostly a matter of practice: you don't need to be a genius; you just have to be willing to work a little. As you gain experience, you'll find yourself developing a sense of what will work and what won't; the blind alleys will become less common.

Having said that, here are some suggestions for further practice:

- One criticism of the word-counting program might be that it destroys the input line; what if the program needed to use the line for something else? Rewrite the program to count the words in the line without modifying the string.
- Write a program that deletes all blanks from an input string.
- Write a program that replaces all occurrences of the character e in an input string with the character f.
- Write a program that will substitute any given character for any other given character in an input string.

(Have your program accept the replaced and replacement characters as input.)

- Write a program to replace string-1 with string-2 everywhere it occurs in string-3. All three strings should be accepted as input by the program. (Do you have to assume string-1 and string-2 are different?)
- Write a program to count the number of occurrences of a given character in a string.

MORE MACINTOSH PASCAL FUNCTIONS

There are a number of additional functions provided by Macintosh Pascal. The functions we will discuss are summarized in Table 7-5.

A good place to start in our discussion is the *random* function, which we have used without explaining in Chapters 4 and 6. The random function has no argument list; when it is used in a program, it looks just like an undeclared variable. The random function call always represents an unpredictable integer value in the range −32768 to 32767. In theory, any value in this range is equally likely to occur. The following program demonstrates how random works by simply printing its value ten times; try it and verify that you obtain different values each time the program runs:

```
program random_lab;
{ experiments with random function }

 var
  i : integer;

begin { random_lab }
 for i := 1 to 10 do
  writeln(random)
end.
```

The random function can be used in any program you want to behave differently each time it is run, typically

Table 7-5.

Additional Macintosh Pascal Functions	
button	
arguments:	none
result type:	Boolean
description:	TRUE if Mouse button is currently down, else FALSE
random	
arguments:	none
result type:	long integer
description:	unpredictable value in the range −32768 to 32767
tickcount	
arguments:	none
result type:	long integer
description:	number of jiffies since Macintosh was started

games or simulation programs. Usually, the **mod** operator can be used with the random function call to obtain random integers in whatever range you want. For example, in a program involving dice throws, the statement

```
die := 1 + random mod 6
```

will assign a random integer value in the range 1 to 6 to the variable die. In general, when you desire an unpredictable integer value in the range low_val to high_val, the expression

```
low_val + random mod (high_val - low_val + 1)
```

will often do an adequate job.

You may also use random to obtain unpredictable values of other types. We saw in Chapter 6 where the expression

```
chr(random mod 26 + ord('a'))
```

was used to obtain a random lowercase letter. Now that you know about the chr and ord functions, you should be able to see why it works. As another example, the following program prints out random four-letter words.

```
program random_lab;
{ experiments with random function }

 var
  i : integer;

begin { random_lab }
 while TRUE do
  begin
   for i := 1 to 4 do
    write(chr(random mod 26 + ord('a')));
   write(' ')
  end
end.
```

Try to see how long it takes for this program to come up with a word that's recognizable. (It's surprising to see how few random words are even slightly pronounceable, let alone actual words.) An easy modification to this program will result in its printing random words of differing lengths. (Hint: choose a random number in, say, the range 1 to 10 for each word's length.)

How would you obtain random characters in other ranges besides a to z? How would you go about calculating random real numbers between 0 and 1? Between any two real values? How would you obtain a random Boolean value? (The latter would be useful for simulating an unpredictable yes-or-no decision.)

Another function you might find useful in your own programs is the *button* function. This simple but powerful function returns a Boolean value of TRUE if the Macintosh's Mouse button is currently being pressed or FALSE if the button is not currently being pressed. The following program shows its use:

```
program button_lab;
{ experiments with the button function }

begin { button_lab }
 while TRUE do
  begin
   writeln('The button is up...');
   while not button do
```

```
    ;
    writeln('The button is down...');
    while button do
  end
end.
```

Type in this program and run it. Press and release the Mouse button a few times. As you might have suspected, it displays the message

```
The button is down...
```

when the button is depressed and

```
The button is up...
```

when the button is released. You will have to stop the program by choosing the Halt option from the Pause menu.

This program, while perhaps not very interesting in itself, contains two statements you will see over and over in Macintosh Pascal programs. The first,

```
while not button do
```

causes the program to wait for the Mouse button to be pressed. The second,

```
while button do
```

causes the program to wait for the button to be released. Note each **while** statement contains no statements inside the loop. (Or more precisely, each contains a Pascal empty statement inside the loop.)

Another function that is easy to use to good effect in your own programs is the *tickcount* function. Tickcount simply returns the number of *jiffies* since the Macintosh was started. (One jiffy is equal to 1/60 second). The value returned from a single call to tickcount is usually not interesting in itself; usually it is combined with a value returned from a previous call to tickcount. For example, one might modify the button_lab program with calls to tickcount that allow the program to time how long the Mouse button was depressed.

```
program button_lab;
{ experiments with the button function }

var
 tc1, tc2 : longint;

begin { button_lab }
 writeln('Press button to start timing...');
 while not button do
  ;
 tc1 := tickcount;
 while button do
  ;
 tc2 := tickcount;
 writeln('Button held for ', (tc2 - tc1) / 60.0 : 1 : 2, ' seconds.')
end.
```

This turns the Macintosh into a relatively expensive stopwatch. (Modify the program, if you want, to continuously display the elapsed time while the button is pressed.)

The tickcount function can also be used to insert precisely timed delays into your program. For example, the following program prints out its initial message, pauses 10 seconds (600 jiffies), and prints a final message:

```
program tick_lab;
{ experiments with the tickcount function }

const
 DELAY = 10;

var
 tc1, tc2 : longint;

begin { tick_lab }
 writeln('I''ll be back in ', DELAY : 1, ' seconds...');
 tc1 := tickcount;
 while tickcount - tc1 < DELAY * 60 do
  ;
 writeln('I''m back!')
end.
```

A simple reaction-time test can be programmed using the tools provided here. For example, an "unexpected" prompt

could be programmed to be displayed after a random delay of, say, 3 to 8 seconds. The time it takes to press the button after the prompt is displayed could then be measured. Type in and run this program:

```
program reaction;
{ test your reaction time }

var
 tc1, tc2 : longint;
 delay : longint;

begin { reaction }
 write('This program tests your reaction');
 write('time. Wait for the word NOW to be ');
 write('printed, then press the mouse ');
 write('button as quickly as you can. Click ');
 writeln('the mouse button once to begin.');
 while not button do
  ;
 while button do
  ;
 writeln;
 write('Press...');
 delay := 180 + random mod 300;
 tc1 := tickcount;
 while tickcount - tc1 < delay do
  ;
 writeln('NOW');
 tc1 := tickcount;
 while not button do
  ;
 tc2 := tickcount;
 write('Reaction time: ');
 write((tc2 - tc1) / 60.0 : 1 : 2);
 writeln(' seconds.')
end.
```

You may notice that it's easy to cheat on this program by simply holding the Mouse button down continuously. Can you modify the program to prevent such cheating? (Hint: test if the button is down just before the prompt NOW is printed. If it is, print an error message instead, and restart the delay.)

MACINTOSH PASCAL MEMORY FUNCTIONS

The remainder of the functions we will discuss in this chapter involve the methods Macintosh Pascal uses to store variables in memory. Since this is a relatively advanced topic, you may skip over this discussion if you want. However, we won't assume any special knowledge about computers on your part, so don't be shy about sticking with us. The functions described in this section are summarized in Table 7-6.

We have mentioned that variables in your program are stored in the computer's memory. Have you wondered how much memory the variables actually occupy? Macintosh Pascal offers the *sizeof* function to answer this question. The sizeof function accepts a single variable name as its argument. (Note that this requirement differs from the usual custom that arguments of function calls be expressions.) Sizeof returns the amount of memory that variable takes up, measured in units called *bytes.*

Here is a simple program to find out how much space is consumed by variables of all the types we have considered so far:

```
program sizes;
{ find memory requirements of various types }

var
 b : Boolean;
 ch : char;
 i : integer;
 l : longint;
 r : real;
 d : double;
 e : extended;
 c : computational;
 s : string;

begin { sizes }
 writeln('Size of Boolean = ', sizeof(b) : 1, ' bytes');
 writeln;
 writeln('Size of char = ', sizeof(ch) : 1, ' bytes');
 writeln;
 writeln('Size of int = ', sizeof(i) : 1, ' bytes');
```

Table 7-6.

Macintosh Pascal Memory Functions	
bitand(i1, i2)	
arguments:	i1, i2-integer values
result type:	long integer
description:	bitwise logical AND of i1 and i2
bitnot(i)	
argument:	i-integer value
result type:	long integer
description:	bitwise logical NOT of i
bitor(i1, i2)	
arguments:	i1, i2-integer values
result type:	long integer
description:	bitwise logical OR of i1 and i2
bitshift(i, n)	
arguments:	i, n-integer values
result type:	long integer
description:	value of i shifted (abs(n) **mod** 32) bits; left-shift if n > 0, right-shift if n < 0
bitxor(i1, i2)	
arguments:	i1, i2-integer values
result type:	long integer
description:	bitwise logical XOR (exclusive-or) of i1 and i2
hiword(i)	
argument:	i-integer value
result type:	long integer
description:	upper 16 bits of i
loword(i)	
argument:	i-integer value
result type:	long integer
description:	lower 16 bits of i
sizeof(v)	
arguments:	any variable name (non-file variable)
result type:	long integer
description:	amount of memory used by the named variable

```
    writeln('Size of longint = ', sizeof(l) : 1, ' bytes');
    writeln;
    writeln('Size of real = ', sizeof(r) : 1, ' bytes');
    writeln('Size of double = ', sizeof(d) : 1, ' bytes');
    writeln('Size of computational = ', sizeof(c) : 1, ' bytes');
    writeln('Size of extended = ', sizeof(e) : 1, ' bytes');
    writeln;
    writeln('Size of string = ', sizeof(s) : 1, ' bytes')
  end.
```

The repetitive typing involved in entering this program can be decreased by using copy-and-paste functions. The output should appear as follows:

```
Size of Boolean = 1 bytes

Size of char = 1 bytes

Size of int = 2 bytes
Size of longint = 4 bytes

Size of real = 4 bytes
Size of double = 8 bytes
Size of computational = 8 bytes
Size of extended = 10 bytes

Size of string = 256 bytes
```

You'll notice that the extended range and precision offered by the nonstandard varieties of real and integer types do not come for free. Variables of those types consume twice as much memory (or more) as those of the normal types.

Another important point is that a variable of the normal **string** type occupies a whopping 256 bytes; it's not too hard to see that using more than a few **string** variables in your program can rapidly exhaust the available memory. There is one way to cut down on string-storage requirements by declaring a smaller string size in the variable declaration.

```
var
  ...
s : string[10];
  ...
```

Try this modification and see how it affects the output.

Actually, the byte is just one way to measure memory storage requirements. Just as distances may be measured in miles, feet, inches, or centimeters, computer memory is measured in different units depending mainly on which unit is most convenient. We will use four different units for memory measurement: *bits*, *bytes*, *words*, and *kilobytes*. (The kilobyte unit is often abbreviated as the single ketter K.) There is nothing mysterious about converting from one unit to another;

Table 7-7.

Memory Measurement Conversion Factors

1 Kilobyte = 512 words = 1024 bytes = 8192 bits
1 word = 2 bytes = 16 bits
1 byte = 8 bits

it is as easy as converting feet to inches. All you need to know are the conversion factors: the numbers to multiply or divide by. The conversions between different memory-measurement units are shown in Table 7-7.

Note that you don't have to know anything about what these units actually represent to perform a conversion. For example, if your Macintosh has 128 kilobytes, you can determine that 128K is the same as 128 times 1024, or 131,072 bytes.

The actual amount of memory consumed by a string variable is one more byte than its assumed size. For example, the default string size of 255 characters is 256 bytes. The reason is that, in addition to the string characters themselves, string variables contain a byte that holds a number giving the current length of the string, whatever it is. Normally the only way to access this byte is through the length function, which simply returns its value.

The smallest unit of the computer's memory is the *bit.* As shown in Table 7-7, all the larger units of the computer's memory are simply sequences of bits. By definition, a bit can store only two possible values; these values are usually thought of as 0 and 1. It's important to realize that all the different values of all the different types we've seen so far are simply differing bit sequences held in the computer's memory —that is, just various combinations of 0 and 1.

Macintosh Pascal offers a number of library functions that allow operations on the underlying bit representation of the integer types (as opposed to the mathematical operations on the integer values themselves).

First let's consider splitting a long integer into two pieces. If you examine the output from the sizes program, you'll note the long integer type can be considered to be made up of two normal integers. Integers use 2 bytes (one word) while long integers consume 4 bytes (two words). The *hiword* and *loword* library functions accept a two-word long integer value and split it into two integer-size parts. Hiword returns the higher

(first) half of its long integer argument as a normal integer value, while loword returns the lower (second) half.

Try out hiword and loword in the Observe window; give each one a number of different arguments. Note that, since integer-type expressions are always automatically converted to long integer values, it is acceptable to use normal integer values as arguments to hiword and loword. And since the function calls are themselves integer-type expressions, they should be considered to represent long integer results.

Macintosh Pascal also offers library functions that allow operations on the individual bits contained in integer-type values. The simplest one is the *bitnot* function. It simply returns the result obtained by inverting all the bits in its argument; 1 bits become 0 and 0 bits are turned into 1's. Again, try out bitnot in the Observe window with a few different arguments. You will remember we discussed the idea of inverse functions with chr and ord previously; it turns out that bitnot is its own inverse. Can you see why? Try to verify this using the Observe window.

Macintosh Pascal also provides the *bitshift* function; it returns the result of moving the bits in its first argument left or right by the number of positions specified by its second argument. If the second argument is positive, bits are shifted left; if negative, the shift is to the right. Empty bit positions are filled in with zeros. Once again, try using bitshift in the Observe window. (Restrict yourself to small shift arguments at first.)

The final three bit functions we'll consider are *bitand*, *bitor*, and *bitxor*. Each operates on two integer-valued arguments. Each bit of the result value comes from an operation between the corresponding bits of the argument values:

- For the bitand function, a bit in the result is set to 1 if both corresponding bits from each argument are also 1.
- In the bitor function, a result bit is set to 1 if either or both corresponding argument bits are 1.
- The bitxor operation sets a result bit to 1, if either *but not both* corresponding argument bits are 1.

These bit operations are known as AND, OR, and XOR, respectively. The rules for each operation (as well as the NOT operation used in the bitnot function) are summarized in tabular form in Figure 7-2.

So far, we have avoided discussing the precise way in which integer values are represented by bit sequences in

AND	0	1
0	0	0
1	0	1

XOR	0	1
0	0	1
1	1	0

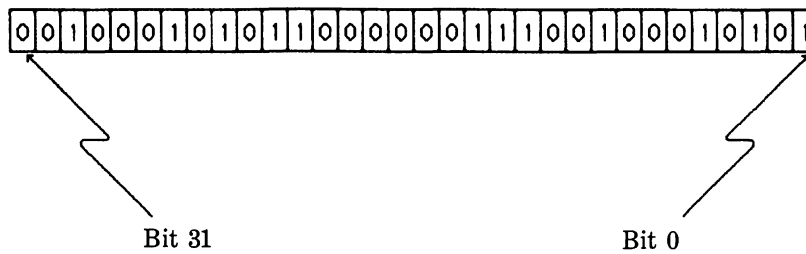
OR	0	1
0	0	1
1	1	1

NOT	0	1
	1	0

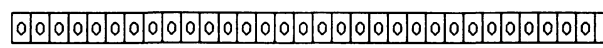
Figure 7-2.

Bit operator tables

memory, but the bit functions allow us to discover how it's done. It is customary to number each bit in a value, calling the rightmost bit bit number 0 and counting to the left. For a 32-bit integer value, this would work as follows:



Let's make the reasonable and correct assumption that an integer value of 1 is simply a sequence of 31 zero bits and a 1 in the rightmost position (bit 0):



Notice that when this value is ANDed with another integer, the outcome will depend only on the rightmost bit of that integer. If the rightmost bit of the other integer is 1, the

result of the AND will be 1. And if the rightmost bit is 0, the result of the AND will be 0. In short, the expression

```
bitand(x, 1)
```

always gives the rightmost bit of x, no matter what x is. Programmers call these kinds of bit patterns *masks*, because they "mask out" all but the desired bits from a value.

Masks can easily be used to find the value of any bit in an integer. To determine bit number i in the integer x, for example, first shift that bit into the rightmost position using a right bitshift of i bits.

```
bitshift(x, -i)
```

Then AND the resulting value with 1 (our mask value):

```
bitand(bitshift(x, -i), 1)
```

That's all there is to it. It's easy, of course, to repeat this 32 times, once for each bit. Putting this strategy into a program might go like this:

```
program bit_lab;
{ print bit representation of input integer }

 var
  x : longint;
  i : integer;

begin { bit_lab }
 write('Enter an integer: ');
 readln(x);
 write('Bit representation of ', x : 1, ':');
 for i := 31 downto 0 do
  write(bitand(bitshift(x, -i), 1) : 1);
 writeln
end.
```

Type in this program and test it on a number of different values.

A simple test program to verify that the bit functions work as expected can be built on this foundation. The following program accepts two integers and calculates the result from XORing them together. The bit representations of all three values are then displayed.

```
program bit_lab;
{ examine bitxor function }

 var
  x, y, z : longint;
  i : integer;

begin { bit_lab }
 write('Enter an integer: ');
 readln(x);
 write('Enter another integer: ');
 readln(y);
 z := bitxor(x, y);
 writeln(x : 1, ' XOR ', y : 1, ' is ', z : 1);
 writeln('Bit representation:');
 write(' ' : 4);
 for i := 31 downto 0 do
  write(bitand(bitshift(x, -i), 1) : 1);
 writeln;
 write('XOR ' : 4);
 for i := 31 downto 0 do
  write(bitand(bitshift(y, -i), 1) : 1);
 writeln;
 write('is ' : 4);
 for i := 31 downto 0 do
  write(bitand(bitshift(z, -i), 1) : 1);
 writeln
end.
```

This demonstration program is useful for understanding what the bitxor function really does; seeing what happens to the bit representations is considerably easier than trying to follow the process by looking at the decimal values. A suggested exercise would be to make slight modifications to this program to allow you to view the results of the other bitwise operations, including hiword and loword.

INTRODUCTION TO LIBRARY PROCEDURES

8

Procedures help make entire programs transparent.

DOUG COOPER AND MICHAEL CLANCY,
Oh! Pascal!
(W.W. Norton, 1982)

In addition to the library functions we discussed in the previous chapter, Macintosh Pascal offers a number of other library subprograms called library *procedures*. Like library functions, library procedures are built into Macintosh Pascal; you may use them in any program you write. Unlike the library functions, however, the library procedures are usually considered to *perform actions* rather than *calculate results*, as we'll see.

STANDARD PROCEDURES

Table 8-1 lists the five standard procedures offered by Macintosh Pascal for simple input and output.

We have been using four of these five standard procedures all along: write, writeln, read, and readln. And, although you may not have realized it, since you know how to use these four, you already know how procedures are called. The syntax for a procedure call is shown in Figure 8-1.

The syntax of a procedure call is the same as that of a function call. A procedure call and a function call differ considerably, however, in *where* they may legally appear in a

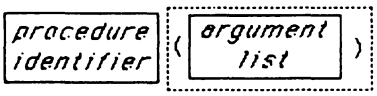


Figure 8-1.

Procedure call syntax

program. Pascal's unshakable rule governing placement of procedure and function calls is as follows:

A function call represents a value;
a procedure call is a statement.

A procedure call may legally appear anywhere in your program that a statement may be placed. A function call, on the other hand, must appear only as an expression or be contained within an expression. So the following examples are totally meaningless in Pascal:

```
y := writeln(x, z);
odd(j)
```

In the first example a procedure call to writeln is being used as a function call; in the second, a call to the function odd is being used as a statement. Both usages are illegal. You must know whether a subprogram is called as a procedure or a function in order to use it in your own programs.

In addition, you must also know what kind of arguments a subprogram expects. Since you are already somewhat familiar with the workings of read/readln and write/writeln, consider the following statement. Is it legal?

```
writeln(2 * 3 + 6)
```

The answer is yes. In general, the write and writeln procedures may be called using expressions as arguments. But does it make any sense to do the same thing with read or readln?

```
readln(2 * 3 + 6)
```

A little thought should tell you that a statement like this is nonsensical. Each argument used in an argument list to read and readln *must* be a variable identifier. Why? Because read and readln argument lists return results—the values of whatever was read. If you do not specify variable names in the argument list, there will be no place to store those returned results. So there are two different kinds of arguments used by procedures and functions: values and variables.

Whenever you write a subprogram call (procedure call or function call) in your own program, you must know two simple things about each argument involved:

- You must know whether the subprogram expects a value or variable argument.
- You must know what type of argument (integer, Boolean, and so on) is expected by the subprogram.

Notice the entries for read and readln in Table 8-1 state

Table 8-1.

Standard Pascal Procedures for Simple Input and Output

page
arguments: none
description: clears Text window

read(r1, r2, ..., rn)
arguments: r1, r2, ..., rn—variables of any readable type (one or more)
description: reads values typed at keyboard into named variables

readln(r1, r2, ..., rn)
arguments: r1, r2, ..., rn—variables of any readable type (zero or more)
description: reads values typed at keyboard into named variables and skips to next input line
note: if zero arguments, parentheses are omitted

write(w1, w2, ..., wn)
arguments: w1, w2, ... wn—any writable values (one or more); each may have formatting parameters
description: prints values in Text window

writeln(w1, w2, ..., wn)
arguments: w1, w2, ... wn—any writable values (zero or more); each may have formatting parameters
description: prints values (if any) in Text window and skips to next output line
note: if zero arguments, parentheses are omitted

the arguments must be variables; the entries for write and writeln indicate that the arguments are values.

In addition to the four procedures already discussed, Table 8-1 mentions the page procedure. It may be used anywhere in your program to erase the Text window. The simple example shown here

```
program ASCII;
{ display ASCII character set }

 var
  i : integer;

begin { ASCII }
 for i := 0 to 255 do
  begin
   write(i : 5, ':', chr(i) : 2);
   if i mod 8 = 7 then
    begin
     writeln;
     if i mod 64 = 63 then
      begin
       writeln;
       writeln('Click mouse to continue...');
       while not button do
        ;
       while button do
        ;
       page
      end
     end
  end
end.
```

is a modification of the ASCII program from Chapter 7 and prints both the characters and their numeric ASCII equivalents. After each line of 8 characters is printed, the program skips to a new line. After each set of 64 characters (eight lines) is printed, the program waits for the Mouse button to be pressed and released. Once that happens, the page procedure is called to clear the window and output continues. (You'll want to expand the Text window to fill the full screen before you run this program. You may also want to find out what happens when the call to page is omitted.)

MACINTOSH PASCAL STRING-MANIPULATION PROCEDURES

The remaining library procedures discussed in this chapter are nonstandard; they may not be present in other versions of Pascal. The first group of procedures we'll consider are two string-manipulation procedures summarized in Table 8-2.

The *insert* procedure inserts one string into another at a specified position. It performs a similar task to the include function discussed in Chapter 7. In fact, any call to the include function that is simply used to insert a string value into a string variable, such as

```
s := include('arg', s, 3)
```

can be replaced with a simpler call to the insert procedure that does exactly the same thing:

```
insert('arg', s, 3)
```

Similarly, a call to the omit function used only to delete characters from a string variable, as in

```
s := omit(s, 3, 2)
```

may be replaced with a call to the *delete* procedure instead:

```
delete(s, 3, 2)
```

Table 8-2.

Macintosh Pascal Procedures for String Manipulation

insert(s1, s2, i)	
arguments:	s1—**string** value
	s2—**string** variable
	i—integer value
description:	inserts s1 into s2 at character #i
delete(s, i, n)	
arguments:	s—**string** variable
	i, n—integer values
description:	deletes n characters from s starting at character #i

The decision as to whether procedures or functions should be used is primarily guided by convenience. The insert and delete procedures are recommended when you want to *change* a string variable. (That's why they demand a string variable as an argument.) The analogous include and omit functions should be used when you merely want to use the *result* of the insertion or deletion and leave the string variables unaffected. As an exercise, return to the last chapter's programs and replace calls to include and omit with calls to insert and delete.

As another example of string manipulation, let's build a palindrome detector. A palindrome is a string that reads the same backwards and forwards, neglecting capitalization and non-letters. Well-known examples are "Able was I ere I saw Elba" and "Madam, I'm Adam."

The strategy we'll try first is to build a "reversed" string from the original, then to compare the reversed string with the original. If they are the same, the original string was a palindrome. As we saw in Chapter 7, our program will accept and process an arbitrary number of lines, stopping when an empty line is entered. An initial pseudo-code might go as follows:

```
get first input string
while input string is not empty
    build reversed string from input string
    compare input string with reversed string
    if equal
        input string is a palindrome
    else
        input string isn't a palindrome
    get next input string
end while
```

Do you see what's wrong with this design? We have forgotten that we must ignore capitalization and non-letters when testing for palindromes. Instead of working with the original string, let's copy the original into a second string variable. We'll then delete all non-letters from this second string and convert all its letters to a single case. Finally, the converted string will be reversed into a third string. If the converted string and its reverse are the same, we have a palindrome. Summarizing this in pseudo-code:

```
get first input string
while input string is not empty
    converted string := input string
    delete all but letters from converted string
       and convert all letters to uppercase
    build "reversed" string from converted string
    compare converted string with reversed string
    if equal
        input string is a palindrome
    else
        input string isn't a palindrome
    get next input string
end while
```

The string processing needed to remove non-letters and convert letters to a single case will be a little tricky. Can we use a normal **for** loop to scan all the characters in the string, as follows?

```
for i := 1 to length(s) do
  ...
```

The answer is no. If we delete characters from the string, the length of the string will change during the execution of the **for** loop. But remember that the upper limit of the **for** is calculated before the loop is entered, so the number of times the loop is executed depends only on the *original* length of the string. If any characters are deleted from the string, the **for** will attempt to access characters beyond the new length of the now shorter string, which will lead to undesired results. It's probably more correct to use a **while** loop and to test the string index variable against the current length of the string each time through the loop. Making this refinement to the pseudo-code gives the following:

```
get first input string
while input string is not empty
    converted string := input string
    i := 1
    while i <= length(converted string) do
        extract character #i from string
        if character is a lowercase letter
            convert to uppercase
            increment i
```

```
        else if character is an uppercase letter
            just increment i
        else (character is not a letter)
            delete character from converted string
        end if
    end while
    build "reversed" string from converted string
    compare converted string with reversed string
    if equal
        input string is a palindrome
    else
        input string isn't a palindrome
    get next input string
end while
```

Note we have also explained the process of deleting non-letters and case conversion in a little more detail. Why is the string index variable i not incremented when a non-letter character is deleted from the string? (Hint: what happens to characters in the string that follow a deleted character?)

Translating this into Pascal gives us the following code:

```
program palindrome;
{ Test input strings for palindromousness }

var
 instr, convert, reverse : string;
 ch : char;
 i : integer;

begin { palindrome }
 write('This program tests strings to see if they are ');
 write('palindromes. Enter each string you want to ');
 write('test on a separate line. Enter an empty ');
 writeln('string when done.');
 writeln;
 readln(instr);
 while length(instr) > 0 do
  begin
   convert := instr;
   i := 1;
   while i <= length(convert) do
    begin
     ch := copy(convert, i, 1);
```

```
    if (ch >= 'a') and (ch <= 'z') then
     begin
      delete(convert, i, 1);
      insert(chr(ord(ch) - ord('a') + ord('A')), convert, i);
      i := i + 1
     end
    else if (ch >= 'A') and (ch <= 'Z') then
     i := i + 1
    else { delete any non-letter }
     delete(convert, i, 1)
   end;
  reverse := '';
  for i := 1 to length(convert) do
   insert(copy(convert, i, 1), reverse, 1);
  write('"', instr, '"');
  if convert = reverse then
   write(' is ')
  else
   write(' isn''t ');
  writeln('a palindrome.');
  readln(instr)
 end
end.
```

Type in and test this program using the palindromes given previously; also make sure it reports non-palindromes correctly. In typing in the program, remember that a null string is typed as two apostrophes, not as a double quote.

SIMPLE QUICKDRAW PROCEDURES

You may remember that we used the Drawing window briefly in Chapter 3 as part of our discussion of Macintosh Pascal cut-and-paste editing. Here is one version of the program from Chapter 3:

```
program quad;
{ Draw a quadrilateral }

begin { quad }
 moveto(50, 100);
 lineto(100, 100);
```

```
    lineto(100, 50);
    lineto(50, 50);
    lineto(50, 100)
  end.
```

Type in and run this program (or retrieve it from disk, if you saved it). In this program we used moveto and lineto to draw simple line figures in the Drawing window. You probably now recognize moveto and lineto as procedure calls.

Moveto and lineto are only two of the many library subprograms provided by Macintosh Pascal that allow your programs to use the Macintosh graphics capabilities. Collectively, these subprograms are known as the *QuickDraw* library, named after the fast, powerful graphics routines contained in the Macintosh's memory. Nearly everything you see on the Macintosh screen (windows, fonts, icons, and so on) is placed there by QuickDraw routines. Full use of QuickDraw involves slightly more knowledge of Pascal than we have covered as yet, but many simple QuickDraw routines can be used with only a little more background. We'll cover those in this chapter's remaining sections.

Fundamental to QuickDraw is the concept of a *coordinate system.* Simply stated, a coordinate system is a method used to translate places or points into numbers, and vice versa. Because the computer only deals with numbers, when you tell the computer to draw a line from here to there, both "here" and "there" must be given as numbers in the coordinate system.

You should be aware that QuickDraw actually uses *many* different coordinate systems to translate places on the Macintosh screen into numbers. Luckily, you only need to know one coordinate system right now in order to use the simple QuickDraw routines.

The coordinate system you'll be using most often is the Drawing window's *local coordinate system.* It is called the local coordinate system because it relates only to the Drawing window. That is, it allows positions to be specified within the Drawing window without having to know where the Drawing window actually is on the screen.

To specify the location of a point within the Drawing window, you need to provide two numbers: the point's horizontal position (how far over from the left it is) and its vertical position (how far down). By custom, the horizontal coordinate is called x and the vertical coordinate y. Positions are measured

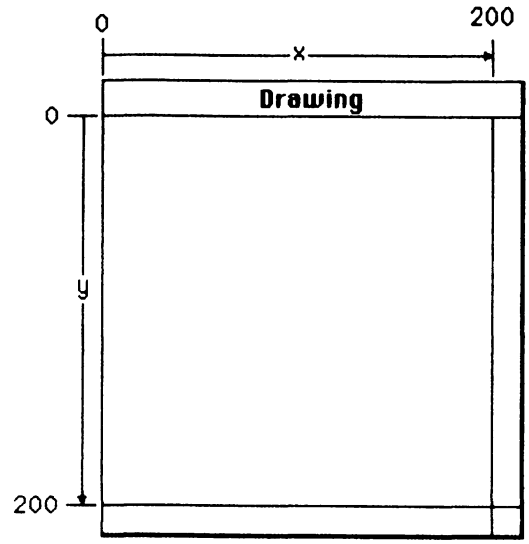


Figure 8-2.

Drawing window local coordinate system

from the upper-left corner of the visible region of the Drawing window.

The units for measuring the horizontal and vertical distances in the coordinate system are *pixels* (which is short for "picture elements"). On the Macintosh screen, the smallest possible black or white dot on the screen is defined to be one pixel wide by one pixel high. The Macintosh screen measures 342 pixels in the vertical direction by 512 pixels in the horizontal direction. Everything you see on the screen is simply differing black and white arrangements of these 175,104 (342 × 512) pixels.

When Macintosh Pascal is started, it sets the interior of the Drawing wintow to a square of 200 by 200 pixels. The Drawing window's coordinate system is shown in Figure 8-2.

To draw in the Drawing window, you specify points as pairs of numbers according to this coordinate system. Think of moving an imaginary pen around inside the Drawing window from point to point; straight lines may be drawn by simply moving the pen from one position in the Drawing window to another.

Table 8-3.

Simple Line Drawing and Pen Movement Procedures

drawline(x1, y1, x2, y2)	
arguments:	x1, y1, x2, y2—integer values
description:	draws line from (x1, y1) to (x2, y2)
line(dx, dy)	
arguments:	dx, dy—integer values
description:	draws line from current pen position (x, y) to (x + dx, y + dy)
lineto(x, y)	
arguments:	x, y—integer values
description:	draws line from current pen position to (x, y)
move(dx, dy)	
arguments:	dx, dy—integer values
description:	moves pen from current pen position (x, y) to (x + dx, y + dy); no drawing is done
moveto(x, y)	
arguments:	x, y—integer values
description:	moves pen to (x, y); no drawing is done

You may also move the pen without line-drawing. Macintosh Pascal provides five built-in procedures to accomplish simple line drawing and pen movement for you. These procedures are summarized in Table 8-3.

We used the moveto and lineto procedures in the quadrilateral-drawing program; you are now in a position to appreciate how they work. For example, the first statement,

```
moveto(50, 100);
```

commands the drawing pen to move to x coordinate 50 and y coordinate 100 without drawing a line. The second statement,

```
lineto(100, 100);
```

tells the pen to move from its current position (x = 50, y = 100) to a new position (x = 100, y = 100), drawing a line as it moves. Figure 8-3 shows the square drawn by the program; each corner is labeled with its coordinates.

If there is the slightest doubt in your mind how the quad program draws a square, study it carefully until you understand it. (Try using the Step option or COMMAND-S to perform the program statements one at a time, observing what each one does.)

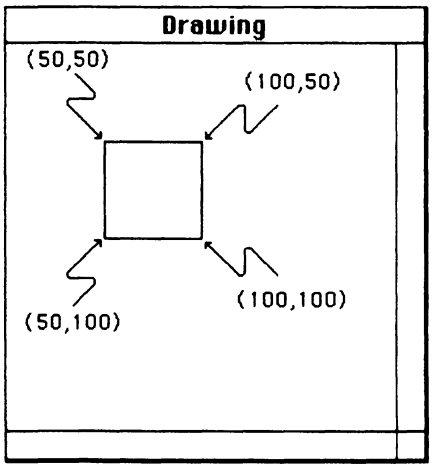


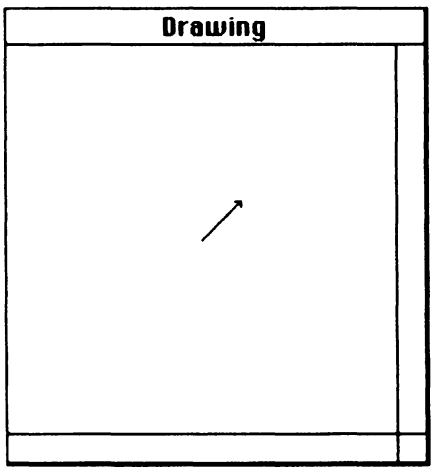
Figure 8-3.

Sample Drawing window coordinates

Complementary to moveto and lineto are the procedures *move* and *line*. Moveto and lineto require actual Drawing window coordinates as their arguments. Move and line, however, use two arguments specifying *relative* pen movements, such as "move *this* far over in the window and *this* far down." The end of the drawn line is relative to the original pen position. Loosely speaking, move and line are useful when it is more convenient to specify the movement of the pen than its actual position. For example, examine this program, which draws a simple arrow.

```
program arrow;
{ draw arrow }

begin { arrow }
 moveto(100, 100);
 line(20, -20);
 line(-3, 0);
 move(3, 0);
 line(0, 3)
end.
```

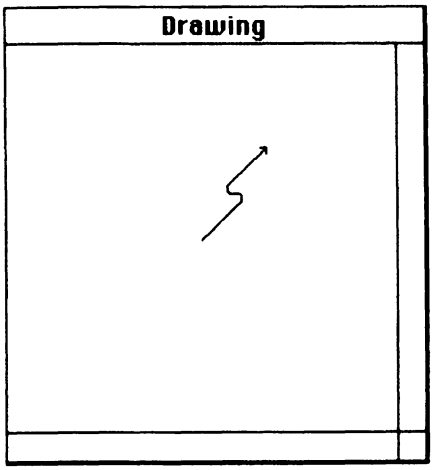
The result of running this program appears as follows:



It's relatively easy to specify a chain of move and line procedure calls to draw simple figures. For example, adding a few lines to the arrow-drawing program,

```
program squiggly;
{ draw squiggly arrow }

begin { squiggly }
 moveto(100, 100);
 line(20, -20);
 line(0, -3);
 move(-1, -1);
 line(-5, 0);
 move(-1, -1);
 line(0, -3);
 line(20, -20);
 line(-3, 0);
 move(3, 0);
 line(0, 3)
end.
```

draws a squiggly arrow.



Part of the convenience of using move and line is that the same statements can be used to draw in different positions. Note that the only statement in this program that specifies where the arrow is actually drawn is the first moveto command. If you want to draw an arrow in another position, you need only to change one line, the moveto command. To demonstrate this, let's wrap two nested **for** loops around the program to get a series of squiggly arrows:

```
program squiggly;
{ draw squiggly arrows }

 var
  i, j : integer;

begin { squiggly }
 for i := 1 to 5 do
  for j := 1 to 5 do
   begin
    moveto(30 * i, 30 * j + 30);
    line(20, -20);
    line(0, -3);
    move(-1, -1);
    line(-5, 0);
```

```
    move(-1, -1);
    line(0, -3);
    line(20, -20);
    line(-3, 0);
    move(3, 0);
    line(0, 3)
   end
end.
```

Before you run this version of the program, guess how many arrows will be drawn and approximately where they will appear.

The *drawline* routine shown in Table 8-3 is useful in drawing isolated lines. The single procedure call

```
drawline(x1, y1, x2, y2)
```

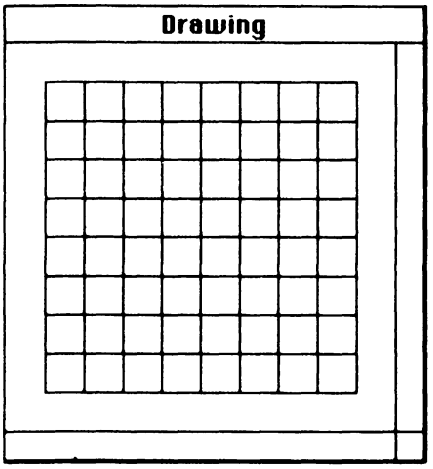
is equivalent to these two procedure calls:

```
moveto(x1, y1);
lineto(x2, y2)
```

As an example of the use of drawline, try the following program for drawing a chessboard:

```
program chessboard;
{ draw a chessboard }

 var
  i : integer;

begin { chessboard }
 for i := 1 to 9 do
  drawline(20, 20 * i, 180, 20 * i);
 for i := 1 to 9 do
  drawline(20 * i, 20, 20 * i, 180)
end.
```

The first **for** loop draws nine equally spaced horizontal lines; the second draws nine equally spaced vertical lines. The net effect is this drawing:



We will return to the subject of line drawing in Chapter 12, where we'll discuss how to change the shape of the pen and the way it draws, among other things.

QUICKDRAW TEXT-DISPLAY PROCEDURES

One of the best features of the Macintosh is its ability to display characters in different sizes and fonts. Characters may be italicized, boldfaced, underlined, outlined, shadowed, or any combination of these. You may have already seen this when using MacWrite or MacPaint. Macintosh Pascal provides you with the power to take advantage of the Macintosh's advanced text-display skills in your own programs. The routines we'll discuss in this section are summarized in Table 8-4.

Displaying text in the Drawing window is quite easy. Try this two-statement program as a first step:

```
program hello;
{ draw a greeting }

begin { hello }
 moveto(5, 100);
 drawstring('Hello there, world!')
end.
```

Table 8-4.

Simple Text-Drawing Procedures

drawchar(ch)
argument: ch—character value
description: draws character ch at current pen position

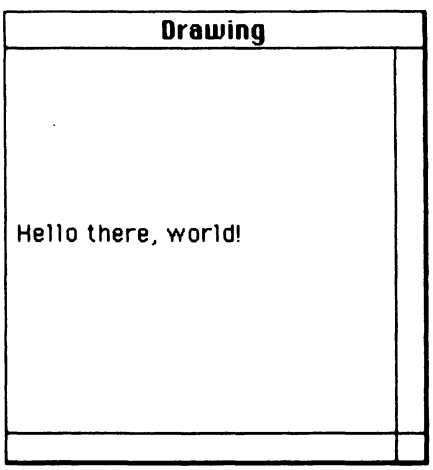
drawstring(s)
argument: s—**string** value
description: draws string s at current pen position

textfont(i)
argument: i—integer value
description: selects font #i for text display

textsize(i)
argument: i—integer value
description: sets font display size to i points

writedraw(w1, w2, ..., wn)
arguments: w1, w2, ..., wn—same as write-arguments, with optional formatting parameters
description: writes formatted values into Drawing window at current pen location

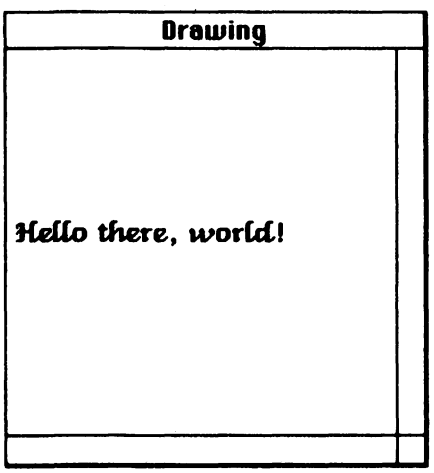
Text will start at the current pen position unless the program calls moveto before calling a text-drawing procedure. We have used drawstring here, which outputs a string to the Drawing window. For convenience, Macintosh Pascal also provides *drawchar*, which can be used to display single characters, and *writedraw*, which writes an arbitrary number of values to the Drawing window, just as write sends its output to the Text window. In our case, the program writes "Hello there, world!" starting at the point (5, 100):



Since we didn't specify which font or character size to use in displaying the text, Macintosh Pascal decided to use the 12-point Geneva font. Changing the font and character size is accomplished by calling the *textfont* and *textsize* procedures. To display our greeting in the decorative 14-point Venice font, modify the program as follows:

```
program hello;
{ draw a greeting }

begin { hello }
 moveto(5, 100);
 textfont(5);
 textsize(14);
 drawstring('Hello there, world!')
end.
```

The result:



You must supply a font number, not a name, to the textfont procedure. The fonts supplied in the System file on the Macintosh Pascal disk are summarized along with their font numbers and available sizes in Table 8-5 (your list of fonts may differ). You may want to use the current program as a base for experimenting with the different fonts and sizes.

If you specify a font number not available in the System file (such as 3), the Geneva font will be used instead. If you specify a font size not provided in the System file, an available font size will be scaled up or down to the desired size.

Table 8-5.

Fonts Provided on Macintosh Pascal Disk

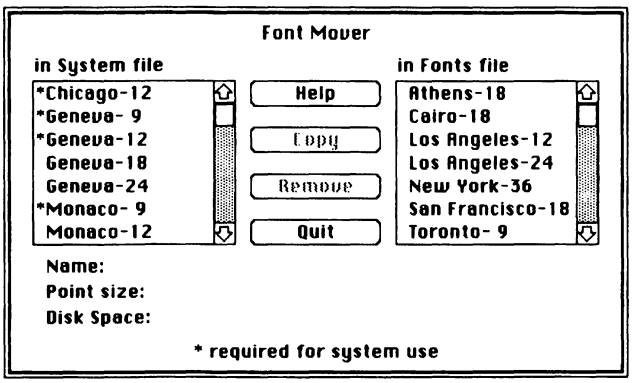
Font Name	Font Number	Available Sizes (points)
Chicago	0	12
Geneva	1	9, 12, 18, 24
New York	2	12
Monaco	4	9, 12
Venice	5	14

This works best when the specified size is a multiple of an available size. Otherwise, the results are often unattractive, to put it mildly.

You may, if you want, install additional fonts (or remove unwanted ones) in the System file using the Font Mover program you received with your Macintosh or with other programs. Only the Chicago, Geneva, and Monaco fonts must remain in the System file. The number of fonts you may move onto the Macintosh Pascal disk is only limited by the amount of free space on the disk.

The fonts you should keep on the Macintosh Pascal disk will depend on the type of programs you write. Here, step by step, is how to install the 18-point Cairo font. (This process assumes you have backed up your original Macintosh Pascal disk using one of the commercially available disk-copying programs and have deleted the demonstration programs and documentation from your copy of the Macintosh Pascal disk, as described in Chapter 3. This provides adequate room on the disk for the Font Mover program and the Fonts file.)

- Copy the Font Mover program and the Fonts data file from a backup copy of the Macintosh System Disk to your working Macintosh Pascal disk.
- Eject any disk from your external disk drive (if you have one). Dispose of all disk icons except the Pascal disk by dragging them to the Trash icon.
- Open the Pascal disk icon (if necessary) and run Font Mover by double-clicking its icon. You should see something like the following dialog box:



- Select the Cairo-18 font by clicking its name in the right-hand box labeled "in Fonts file".
- Click the Copy button to install the font.
- If all went well, click the Quit button to exit from Font Mover.

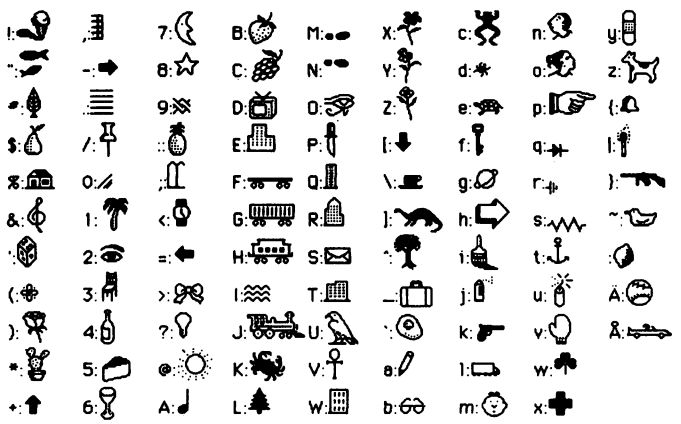
A similar process may be used to move any font onto the Macintosh Pascal disk. After you are done with Font Mover, you may delete both it and the Fonts data file from the disk.

In the Macintosh scheme of things, the Cairo font is number 11. It is a hieroglyphic font; instead of letters and digits and punctuation, it contains a different picture for each character value. The following program can be used to display all the pictures in the Cairo font with their corresponding normal character values:

```
program Cairo;
{ display Cairo font }

 const
  GENEVA = 1;
  CAIRO = 11;

 var
  i, x, y : integer;

begin { Cairo }
 for i := 33 to 129 do
  begin
   x := 5 + 52 * ((i - 33) div 11);
   y := 24 + 26 * ((i - 33) mod 11);
   moveto(x, y);
```

```
    textsize(12);
    textfont(GENEVA);
    writedraw(chr(i), ':');
    textsize(18);
    textfont(CAIRO);
    drawchar(chr(i))
  end
end.
```

The tricky part of this program is adjusting the arguments in the moveto procedure call so that the characters are displayed far enough apart to be easily distinguishable, but close enough together so they will all fit on screen at once. (The complex calculations shown in the program are largely a result of trial and error.) You will need to move and expand the Drawing window so that it fills the entire screen before you run this program. The result should appear as follows:



As was the case with the line-drawing procedures in the previous section, we've given you merely an introduction to the powerful text-display procedures available to your Pascal programs.

SIMPLE SHAPE-DRAWING PROCEDURES

QuickDraw provides many routines other than line drawing and text display for drawing different shapes easily and

Table 8-6.

QuickDraw Shapes and Operations

Shapes	Operations
rectangle	frame
oval	paint
rounded-corner rectangle	erase
arc	invert
polygon*	fill*
region*	

(*not discussed here)

quickly. We will discuss the simple shape-drawing procedures here and leave the more complex ones for Chapter 12.

QuickDraw's shape-drawing procedures are easy to classify. There are six different kinds of shapes and five different drawing operations possible using each shape type. The names of the shapes and operations that apply to them are given in Table 8-6.

There are 30 (6 × 5) possible combinations of shapes and operations, and QuickDraw has built-in procedures available for each combination. We will cover all but one operation here (*filling* will be discussed in Chapter 12).

First, rectangles. Table 8-7 describes the procedures used

Table 8-7.

QuickDraw Rectangle-Drawing Operations

eraserect(top, left, bottom, right)
arguments: top, left, bottom, right—integer values
description: paints interior of specified rectangle using background pattern (normally white).

framerect(top, left, bottom, right)
arguments: top, left, bottom, right—integer values
description: draws frame just inside specifed rectangle using current pen pattern and mode

invertrect(top, left, bottom, right)
arguments: top, left, bottom, right—integer values
description: inverts the color of all pixels enclosed by specified rectangle

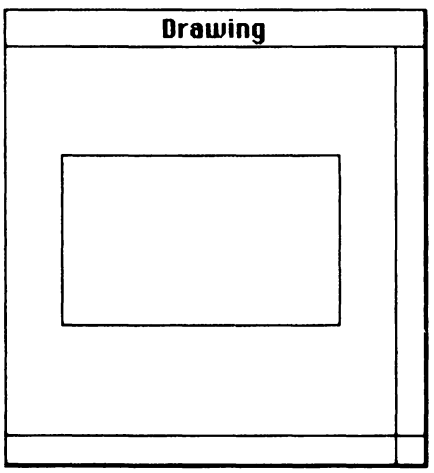
paintrect(top, left, bottom, right)
arguments: top, left, bottom, right—integer values
description: paints interior of specified rectangle using current pen pattern and mode (normally black)

for the simple operations QuickDraw permits on rectangles. (Don't worry about new terms in Table 8-7 such as "pen pattern" and "mode." They're explained in Chapter 12.)

To draw a rectangle requires that you specify the location of each of the four sides by using the Drawing window's local coordinate system. Try out the following one-statement program to get started:

```
program rect_lab;
{ experiments with rectangles }

begin { rect_lab }
 framerect(55, 28, 144, 172)
end.
```

When this program runs you will be rewarded with a rectangle in the Drawing window:



A call to *framerect* causes a rectangle to be drawn in the Drawing window by using the supplied values for the location of each side. The other three routines—*eraserect, invertrect,* and *paintrect*—are included in the following program to show you what they do.

```
program rect_lab;
{ experiments with rectangles }

var
 i : integer;
```

```
begin { rect_lab }
 framerect(55, 28, 144, 172);
 framerect(100, 50, 113, 68);
 paintrect(120, 60, 140, 170);
 paintrect(60, 120, 170, 140);
 eraserect(120, 120, 140, 140);
 eraserect(70, 125, 160, 135);
 invertrect(125, 70, 135, 160);
 for i := 1 to 1000 do
  invertrect(55, 28, 144, 172)
end.
```

Once more, it is a good idea to run this program in Step or Step-Step mode so you can observe the effect of each procedure call.

We now need to get a little more rigorous in our definitions of some of the terminology we've been using. We have up to now blurred the distinction between points and pixels, but the terms really refer to two different concepts. You may think of the QuickDraw coordinate plane as a grid of infinitely thin horizontal and vertical lines each 1/72 inch apart. QuickDraw points are at the intersection of each horizontal and vertical line. Pixels, on the other hand, are the squares lying *between* each adjacent pair of horizontal and vertical lines. The relationship between grid lines, points, and pixels is illustrated in Figure 8-4. In QuickDraw, points are considered to be infinitely small. Pixels, on the other hand, are 1/72-inch-square areas on the Macintosh screen. You never see points; you only see pixels.

The QuickDraw coordinate system designates points, not

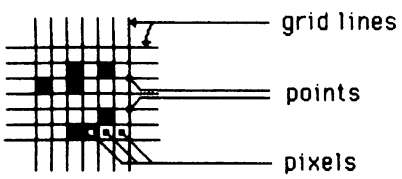


Figure 8-4.

QuickDraw's coordinate grid lines, points, and pixels

pixels. You may think of the horizontal and vertical grid lines as numbered consecutively; the vertical grid lines represent different x coordinates and the horizontal grid lines represent different y coordinates. A coordinate point (x, y) refers to the intersection of vertical grid line x and horizontal grid line y.

A QuickDraw rectangle is specified by the y values of its top and bottom grid lines and the x values of its left and right grid lines. Nearly all procedures that accept four integer values for rectangle coordinates do so in the same order as for the rectangle-manipulation procedures we've just seen: top, left, bottom, right. Precisely speaking, a call to the framerect procedure doesn't exactly draw a rectangle; instead, it draws the pixels that lie *just inside* the boundaries of the rectangle. This is illustrated in Figure 8-5.

All this discussion of points, grid lines, pixels, and rectangles may seem needlessly formal to you. But one of the secrets of creating high-quality graphics on the Macintosh is to have a good grasp on the precise effect of the QuickDraw routines. A little attention to detail at this point will allow your further explorations of QuickDraw to proceed more easily. You'll also find that it's easier to create sophisticated displays when you are comfortable with the exact way the QuickDraw routines work. (You may wish to review the language used in Table 8-7 at this point.)

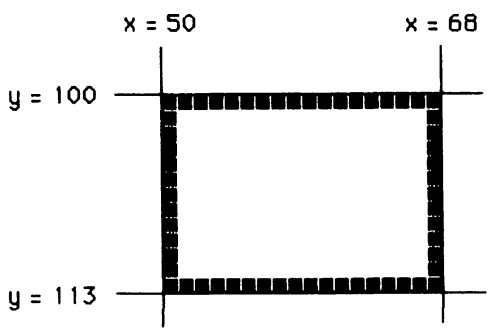


Figure 8-5.

The effect of framerect

A number of flashy effects are possible using our newly acquired routines. For example, the following small program draws another version of a chessboard, this time with alternating white and black squares:

```
program chessboard;
{ draw a chessboard }

 var
  i : integer;

begin { chessboard }
 for i := 1 to 8 do
  invertrect(20 * i, 20 * i, 180, 180);
 for i := 2 to 8 do
  invertrect(20, 20, 20 * i, 20 * i);
 framerect(19, 19, 181, 181)
end.
```

You'll see how this program works if you try running it in slow motion (use Step or Step-Step).

As another example, here is a program that draws a piano keyboard in the Drawing window. Move and stretch the Drawing window to its maximum width before you try running it:

```
program keyboard;
{ Draw a piano keyboard }

 const
  WHITE_WID = 9;
  BLACK_WID = 4;
  WHITE_LEN = 30;
  BLACK_LEN = 20;
  XORG = 10;
  YORG = 10;

 var
  i, x, yw, yb, bwid2 : integer;

begin { keyboard }
 x := XORG;
```

```
  yw := YORG + WHITE_LEN;
  yb := YORG + BLACK_LEN;
  bwid2 := BLACK_WID div 2;
  for i := 1 to 52 do
   begin
    framerect(YORG, x, yw, x + WHITE_WID + 1);
    x := x + WHITE_WID;
    if (i mod 7 <> 2) and (i mod 7 <> 5) and (i <> 52) then
     paintrect(YORG, x - bwid2, yb, x + bwid2 + 1)
   end
end.
```

In this program the **for** loop draws each white key; then the complex **if** test inside the loop decides whether to place a black key to the right of the white key just drawn. (The pattern of black keys repeats every seven white keys, which accounts for the **i mode** 7 calculation in the **if**-test.) For efficiency's sake, however, a number of calculations are done "outside the loop" before the **for** starts.

Once you understand how QuickDraw rectangles work, the remainder of the simple shapes are easy. Let's consider ovals next. Table 8-8 shows the procedure calls for simple oval drawing.

Ovals, like rectangles, are defined by the x and y coordi-

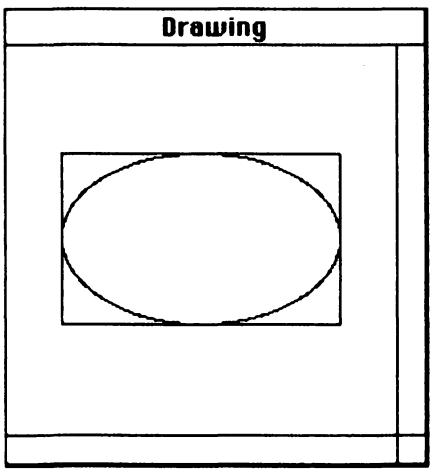
Table 8-8.

QuickDraw Oval-Drawing Operations	
eraseoval(top, left, bottom, right)	
arguments:	top, left, bottom, right—integer values
description:	paints interior of specified oval using background pattern (normally white)
frameoval(top, left, bottom, right)	
arguments:	top, left, bottom, right—integer values
description:	draws outline just inside specifed oval using current pen pattern and mode
invertoval(top, left, bottom, right)	
arguments:	top, left, bottom, right—integer values
description:	inverts all pixels enclosed by specified oval
paintoval(top, left, bottom, right)	
arguments:	top, left, bottom, right—integer values
description:	paints interior of specified oval using current pen pattern and mode (normally black)

nates of their top, left, bottom, and right edges. You can demonstrate this quite easily to yourself with a simple two-line program:

```
program oval_lab;
{ experiments with ovals }

begin { oval_lab }
 framerect(55, 28, 144, 172);
 frameoval(55, 28, 144, 172)
end.
```

When you run this, the result is a rectangle with an inscribed oval:



Like rectangles, QuickDraw ovals should be considered infinitely thin; the framing operation doesn't draw the oval itself, but rather the pixels inside the oval.

Circles are special cases of ovals, just as squares are special cases of rectangles. In QuickDraw, a circle is an oval whose defining rectangle is a square. It is usually more convenient, however, to specify a circle by the position of its center and its radius than the positions of its "sides." For this reason, Macintosh Pascal provides special painting and inverting routines useful for circle drawing, shown in Table 8-9. (If you want to frame, erase, or fill a circle, you'll have to resort to the oval routines.)

Table 8-9.

QuickDraw Circle-Drawing Operations	
invertcircle(x, y, r)	
arguments:	x, y, r: integer values
description:	inverts pixels inside circle centered at (x, y) with radius r
paintcircle(x, y, r)	
arguments:	x, y, r: integer values
description:	paints interior of circle centered at (x, y) with radius r

A simple example shows how *invertcircle* can be used to draw a bull's-eye of alternate concentric bands of black and white:

```
program bullseye;
{ draw a bullseye }

const
 XCENTER = 100;
 YCENTER = 100;
 BANDWIDTH = 20;

var
 r : integer;

begin { bullseye }
 r := 90;
 while r > 0 do
  begin
   invertcircle(XCENTER, YCENTER, r);
   r := r - BANDWIDTH
  end
end.
```

After you run this program, experiment with different values for the constant BANDWIDTH. (Values of 1 and 2 give spectacular results.)

A third shape is the rounded-corner rectangle. Simple drawing routines for this shape are summarized in Table 8-10. The first four arguments to the rounded-corner rectangle routines give the positions of the left, top, right, and bottom edges. The roundness of the corners—each one-quarter of an

Table 8-10.

QuickDraw Rounded-Corner Rectangle Operations

eraseroundrect(top, left, bottom, right, oval_wid, oval_ht)
arguments: top, left, bottom, right, oval_wid, oval_ht—integer values
description: paints interior of specified rounded-corner rectangle using background pattern (normally white)

frameroundrect(top, left, bottom, right, oval_wid, oval_ht)
arguments: top, left, bottom, right, oval_wid, oval_ht—integer values
description: draws frame just inside specified rounded-corner rectangle using current pen pattern and mode

invertroundrect(top, left, bottom, right, oval_wid, oval_ht)
arguments: top, left, bottom, right, oval_wid, oval_ht—integer values
description: inverts all pixels enclosed by specified rounded-corner rectangle

paintroundrect(top, left, bottom, right, oval_wid, oval_ht)
arguments: top, left, bottom, right, oval_wid, oval_ht—integer values
description: paints interior of specified rounded-corner rectangle using current pen pattern and mode (normally black)

oval—is defined by the final two arguments, which give the width and height of that oval. The meaning of each parameter is diagrammed in Figure 8-6.

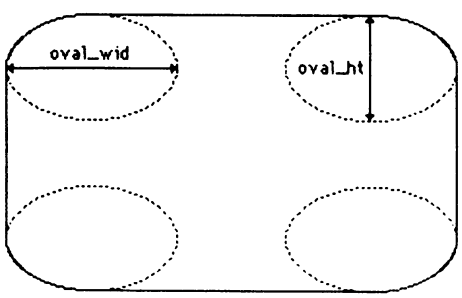


Figure 8-6.

Rounded-corner rectangle geometry

The following program draws a rounded-corner rectangle together with the corner-rounding ovals; you can experiment with the constant declarations to discover the different effects possible. You may be especially interested in finding out what happens when the rounding oval and the rectangle have the same width or height. Try this program now:

```
program round_rect_lab;
{ experiments with rounded-corner rectangles }

 const
  LEFT = 28;
  RIGHT = 172;
  TOP = 55;
  BOT = 144;
  HEIGHT = 34;
  WIDTH = 55;

begin { round_rect_lab }
 frameroundrect(TOP, LEFT, BOT, RIGHT, WIDTH, HEIGHT);
 frameoval(TOP, LEFT, TOP + HEIGHT, LEFT + WIDTH);
 frameoval(TOP, RIGHT - WIDTH, TOP + HEIGHT, RIGHT);
 frameoval(BOT - HEIGHT, LEFT, BOT, LEFT + WIDTH);
 frameoval(BOT - HEIGHT, RIGHT - WIDTH, BOT, RIGHT)
end.
```

The final shape we'll discuss in this chapter is the arc, whose drawing routines are summarized in Table 8-11. An arc is simply a part of the boundary of an oval. Often an arc-drawing operation will affect an entire *wedge*, that is a pie-shaped area that has its apex at the oval's center and expands out to an arc of the oval. When an arc is painted, for example, the entire wedge corresponding to the arc is painted as well.

The arc-drawing routines accept six parameters: the first four define the oval to be used, and the final two specify how much of the oval to use. Positions on the oval—where the arc begins and ends—are measured in degrees. Think of the screen as a map; north (up) is 0°, east (right) is 90°, south is 180°, and west is 270°. The angle-measurement scheme is not a true geometrical angle; intermediate angles depend on the rectangle that encloses the oval. A line from the center of the rectangle through the northeast corner is defined to be at 45°.

Table 8-11.

QuickDraw Arc-Drawing Operations

erasearc(top, left, bottom, right, start_angle, arc_angle)
arguments: top, left, bottom, right, start_angle, arc_angle—integer values
description: paints interior of specified arc using background pattern (normally white)

framearc(top, left, bottom, right, start_angle, arc_angle)
arguments: top, left, bottom, right, start_angle, arc_angle—integer values
description: draws frame just inside specified arc using current pen pattern and mode

invertarc(top, left, bottom, right, start_angle, arc_angle)
arguments: top, left, bottom, right, start_angle, arc_angle—integer values
description: inverts all pixels enclosed by specified arc

paintarc(top, left, bottom, right, start_angle, arc_angle)
arguments: top, left, bottom, right, startangle, arcangle—integer values
description: paints interior of specified arc using current pen pattern and mode (normally black)

Likewise, the southeast corner is 135°, the southwest corner is 225°, and the northwest corner is 315°. (Note this is a true geometrical angle if and only if the defining oval is a circle.) This angle-measurement method is diagrammed in Figure 8-7.

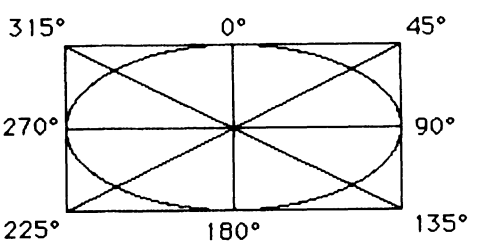


Figure 8-7.

QuickDraw arc measurement

Table 8-12.

Your Company

Income Source	Percentage of Income
Software	30%
Semiconductors	10%
Telecommunications	25%
Children's Toys	35%

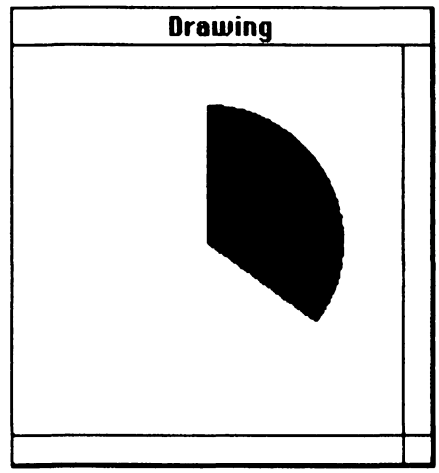
The first angular parameter (start_angle) to the arc-drawing routines specifies where on the oval the arc is to begin, following the scheme just described. The second angular parameter (arc_angle) tells how long the arc is, measured in degrees. The measurement is clockwise if the angle is positive, counterclockwise if negative.

A natural application for arc and wedge drawing is construction of pie charts. Instead of spending a few hundred dollars for commercial software that produces pie charts, you now have the tools to produce your own. Suppose you want to produce a pie chart that reflects the importance of various sources of income enjoyed by your high-tech company shown in Table 8-12.

The percentages shown may be converted into angles by multiplying by 360° (one full circle). The first percentage is then 35% × 360° = 126°. Let's use the *paintarc* procedure to draw the wedge corresponding to 126°. Enter and run this one-statement program:

```
program piechart;
{ draw a pie chart }

 const
  TOP = 30;
  LEFT = 30;
  BOT = 170;
  RIGHT = 170;

begin { piechart }
 paintarc(TOP, LEFT, BOT, RIGHT, 0, 126)
end.
```

The result:



The remaining segments can be drawn in alternate black and white by using the *invertarc* procedure call. Add the following statements just before **end** and rerun the program (don't forget any required semicolons):

```
invertarc(TOP, LEFT, BOT, RIGHT, 0, 234);
invertarc(TOP, LEFT, BOT, RIGHT, 0, 324);
invertarc(TOP, LEFT, BOT, RIGHT, 0, 360)
```

You may want to frame the entire pie with an enclosing circle; to do this, add the following line:

```
framearc(TOP, LEFT, BOT, RIGHT, 0, 360);
```

Notice that the first piece's color has changed from black to white. Why? (Hint: what if there were five pieces?)

You may have seen pie charts with one piece of pie offset from the others for emphasis. This is also rather easy to do, since the position of any wedge is governed by the position of the defining oval; moving the oval moves the wedge, but doesn't change its shape. Let's offset the 126° wedge by first erasing it with *erasearc* and then repainting it slightly north-

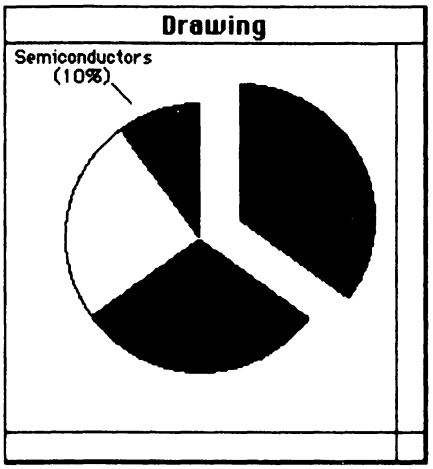
east of its old position. Add the following statements just before **end** and rerun the program:

```
erasearc(TOP, LEFT, BOT, RIGHT, 0, 126);
paintarc(TOP - 10, LEFT + 20, BOT - 10, RIGHT + 20, 0, 126);
```

For most purposes, you'll want to label the pie chart with the appropriate labels and amounts denoted by each slice. Locating the text in the appropriate places is usually done by trial and error. Here is a version of the program that labels one slice:

```
program piechart;
{ draw a pie chart }

 const
  LEFT = 30;
  TOP = 30;
  BOT = 170;
  RIGHT = 170;

 begin { piechart }
  paintarc(TOP, LEFT, BOT, RIGHT, 0, 126);
  invertarc(TOP, LEFT, BOT, RIGHT, 0, 234);
  invertarc(TOP, LEFT, BOT, RIGHT, 0, 324);
  invertarc(TOP, LEFT, BOT, RIGHT, 0, 360);
  framearc(TOP, LEFT, BOT, RIGHT, 0, 360);
  erasearc(TOP, LEFT, BOT, RIGHT, 0, 126);
  paintarc(TOP - 10, LEFT + 20, BOT - 10, RIGHT + 20, 0, 126);
  textsize(9);
  moveto(5, 10);
  drawstring('Semiconductors');
  moveto(25, 20);
  drawstring('(10%)');
  line(10, 10)
 end.
```

The chart produced by this version of the program is as follows:



Labeling the other three slices is left as an exercise for you. Feel free to expand the Drawing window to get a little more room if needed. You may also want to experiment with using an oval pie instead of a circle for a three-dimensional effect.

Even the relatively simple tools discussed in this section can be used to quickly create drawings of recognizable objects and animation effects. Even if you have limited artistic ability, you'll find it easy to put together bits and pieces of programs to create interesting pictures. Here's one simple example:

```
program self_portrait;
{ a mystery program }

var
 i, j : integer;

begin
 frameroundrect(40, 50, 190, 150, 10, 10);
 frameroundrect(40, 50, 170, 150, 10, 10);
 frameroundrect(54, 60, 120, 140, 10, 10);
 framerect(140, 120, 150, 138);
 paintrect(143, 100, 147, 135);
 frameoval(56, 69, 118, 131);
 while TRUE do
```

```
begin
 for i := 1 to 360 do
  paintarc(57, 70, 117, 130, 0, i);
 for i := 1 to 360 do
  erasearc(57, 70, 117, 130, 0, i)
 end
end.
```

ADDITIONAL MACINTOSH PASCAL LIBRARY PROCEDURES

Some very useful Macintosh Pascal procedures don't readily fit into one category. These procedures are listed in Table 8-13.

Table 8-13.

Additional Macintosh Pascal Library Procedures	
getmouse(x, y)	
arguments:	x, y—integer variables
description:	returns current position of mouse (Drawing window local coordinate system used)
getsoundvol(v)	
argument:	v—integer variable
description:	returns current sound volume level (0—silent, 7—loudest)
note(frequency, amplitude, duration)	
arguments:	frequency—long integer value (12 Hz to 783360 Hz) amplitude—integer value (0 to 255) duration—integer value (0 to 255 jiffies)
description:	generates square-wave tone
setsoundvol(v)	
argument:	v—integer value
description:	sets sound volume level (0—silent, 7—loudest)
synch	
arguments:	none
description:	synchronizes program action with screen-drawing cycle
sysbeep(d)	
argument:	d—integer value
description:	generates square-wave tone of approximate duration d * 0.022 seconds

Probably the most useful of these procedures is *getmouse*, which tells the location of the pointer on the screen. The procedure call requires two variable arguments, which contain the x and y coordinates of the pointer when the procedure returns. The coordinate system is the Drawing window's local coordinate system, as usual.

The getmouse procedure can be used to track movements of the Macintosh mouse. For example, the following program draws a line in the Drawing window following the pointer as long as the Mouse button is depressed:

```
program scribble;
{ scribble in the Drawing window }

 var
  mousex, mousey : integer;

begin { scribble }
 while TRUE do
  begin
   while not button do
    ;
   getmouse(mousex, mousey);
   moveto(mousex, mousey);
   while button do
    begin
     getmouse(mousex, mousey);
     lineto(mousex, mousey)
    end
  end
end.
```

You have already used the *sysbeep* procedure in an earlier chapter; it simply generates a square-wave tone similar to the one the Macintosh makes when you turn it on. The integer argument passed to sysbeep controls the duration of the sound; to find the approximate duration in seconds, multiply by the factor 0.022.

You have considerably more control over the Macintosh's sound generator when you use the *note* procedure. Note accepts three arguments; the first is a long integer in the range of 12 to 783360, which is the frequency of the desired note in Hertz. The second is an integer in the range of 0 to 255 and is the desired volume of the note, and the third is an

integer in the range of 0 to 255, which is the duration of the note in jiffies (1 jiffy = 1/60 second).

A little musical knowledge is helpful in putting note to work. On a piano, the lowest key has a frequency of 27.5 Hertz and each successive key's frequency is the twelfth root of two times the frequency of the previous key. The following program plays all 88 keys from the lowest to the highest; note each frequency is first calculated as a real value and then rounded to an integer when it is passed to the note procedure.

```
program keysounds;
{ play piano keys }

const
 LOWNOTE = 27.5;

var
 i : integer;
 freq, ratio : extended;

begin { keysounds }
 ratio := exp(ln(2.0) / 12.0);
 freq := LOWNOTE;
 for i := 1 to 88 do
  begin
   note(round(freq), 255, 5);
   freq := freq * ratio
  end
end.
```

Try this program and you'll discover that very low frequencies aren't handled too well.

If you can plink out one-fingered tunes on a piano, you can do the same on your Macintosh. For example, try the following program:

```
program dragnet;
{ dum de dum dum }

begin { dragnet }
 note(131, 255, 30);
 note(147, 255, 15);
 note(156, 255, 30);
 note(131, 255, 30);
end.
```

The *getsoundvol* and *setsoundvol* procedures are useful when you want your program to generate sound of a certain volume independent of the volume setting on the Macintosh Control Panel. It is usually a good idea to restore the volume to its previous value after you are done. The following example tests the volume at its loudest setting.

```
program dragnet;
{ dum de dum dum }

var
 volume : integer;

begin { dragnet }
 getsoundvol(volume);
 setsoundvol(7);
 note(131, 255, 30);
 note(147, 255, 15);
 note(156, 255, 30);
 note(131, 255, 30);
 setsoundvol(volume)
end.
```

The final procedure we'll discuss here is *synch.* The images you see on the Macintosh screen are put there by an electron gun that "paints" each pixel, and the entire video image is redrawn once every 1/60 second, approximately. When objects on the screen are being changed rapidly, you will often get an interference effect between the rate at which objects are being drawn and the revision rate of the screen. The results are so-called "scanning bars" and excessive flicker. You can see an example by entering and running this program:

```
program scanning_bar;
{ examine scanning bars }

var
 i : integer;

begin { scanning_bar }
 paintrect(50, 50, 150, 150);
 while TRUE do
  begin
```

```
  for i := 1 to 50 do
   begin
    eraserect(50, 49 + i, 150, 151 - i);
    paintrect(50, 50 + i, 150, 150 - i)
   end;
  for i := 50 downto 1 do
   begin
    eraserect(50, 49 + i, 150, 151 - i);
    paintrect(50, 50 + i, 150, 150 - i)
   end
 end
end.
```

After you have observed the behavior of this program, halt it and insert four calls to the synch procedure before each QuickDraw call inside the **for** statements:

```
program scanning_bar;
{ examine scanning bars }

 var
  i : integer;

begin { scanning_bar }
 paintrect(50, 50, 150, 150);
 while TRUE do
  begin
   for i := 1 to 50 do
    begin
     synch;
     eraserect(50, 49 + i, 150, 151 - i);
     synch;
     paintrect(50, 50 + i, 150, 150 - i)
    end;
   for i := 50 downto 1 do
    begin
     synch;
     eraserect(50, 49 + i, 150, 151 - i);
     synch;
     paintrect(50, 50 + i, 150, 150 - i)
    end
  end
end.
```

When you run this version of the program the animation appears much smoother, as desired.

The true test of your understanding of the procedures and functions that Macintosh Pascal gives you is to create some of your own. The next chapter will take you through the process of custom-building your own subprograms.

YOUR OWN PROCEDURES AND FUNCTIONS

9

Man is a tool-making animal.

BENJAMIN FRANKLIN

The convenience of calling subprograms to accomplish complex tasks should be evident by now. In addition to using all the procedures and functions given to you by Macintosh Pascal, you have the option of writing subprograms of your own invention.

WRITING PROCEDURES

In this section we'll discuss how to write procedures that can be called the same way as any library procedures. As you work through the discussion in this section, please keep in mind that many of the points we make concerning procedures apply to functions as well.

Assume, for the moment, that Macintosh Pascal provided a built-in library procedure called put_string that would place a specified string in the Drawing window at a certain position, using a given font and size. We might describe such a library procedure as shown in Figure 9-1. A simple program using this procedure might be written this way:

put_strings(s, x, y, f, p)

arguments: s—**string** value
x, y, f, p—integer values

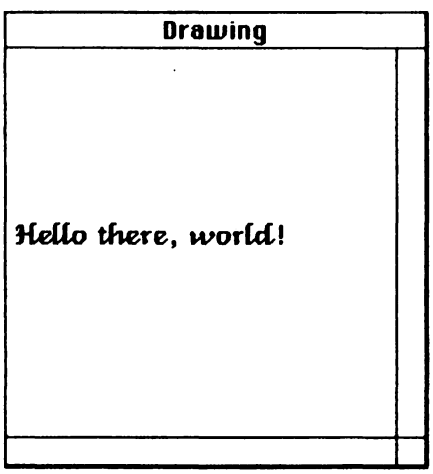
description: displays string s in font f, size p, at coordinates (x, y) in Drawing window

Figure 9-1.

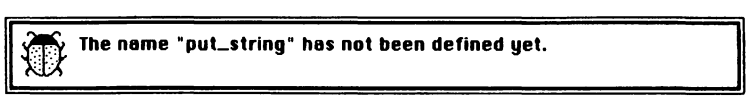
Description of put_string procedure

```
program hello;
{ say hello }

begin { hello }
 put_string('Hello there, world!', 5, 100, 5, 14)
end.
```

This would place the string "Hello there, world!" at Drawing window coordinates (5, 100) using font 5 (Venice) in its 14-point size:



If you type in and try to run this program, however, you get a bug box:



The problem is, of course, that there is no library procedure put_string; trying to call a non-existent procedure is illegal. (You have seen the same problem in trying to use undefined types and variables.)

Even though Macintosh Pascal doesn't provide a put_string library procedure, we already know how to perform the operations such a procedure would have to carry out. In the previous chapter we wrote the string "Hello there, world!" at (5, 100) in 14-point Venice font by using the following:

```
textfont(5);
textsize(14);
moveto(5, 100);
drawstring('Hello there, world!')
```

You should have little trouble abstracting this slightly: to write a **string** variable s at position (x,y) in the Drawing window using font number f and a size of p points, you could use the following statements:

```
textfont(f);
textsize(p);
moveto(x, y);
drawstring(s)
```

A Macintosh Pascal procedure to perform these operations would be written this way:

```
procedure put_string (s : string;
      x, y, f, p : integer);
 { display string s in font f, size p, at (x, y) }

begin { put_string }
 textfont(f);
 textsize(p);
 moveto(x, y);
 drawstring(s)
end;
```

First, and most important, you should notice that this procedure resembles a program: a sequence of statements appears between **begin** and **end**, and the procedure's name (put_string) is placed after the word **procedure**, just as a program's name is placed after the word **program**. Procedures are, quite literally, "little programs" all to themselves.

Second, you should see some differences between procedures and programs. Consider the *procedure heading:*

```
procedure put_string (s : string;
    x, y, f, p : integer);
```

Informally, the procedure heading contains a list of the procedure's arguments in parentheses following the procedure name. The argument list has four functions:

- It specifies the types of the arguments used in a call to the procedure.
- It specifies the order of the arguments in a call to the procedure.
- It specifies whether the arguments in a call are variables or values.
- It gives the names the procedure will use to refer to the arguments.

Put_string's procedure header says that a legal call to put_string will have five arguments; the first one will be a **string** and the remaining four will be integers. Within the procedure, the string value will be called s; the integer values will be called x, y, f, and p. (All five arguments to put_string are values; we'll discuss how variable arguments are specified later in the chapter.)

Unlike programs, procedures are useless all by themselves. If you tried to type the put_string procedure into Macintosh Pascal and run it, you would get a bug box, stating that the **program** keyword is missing. (There is no exception to the rule that Pascal programs must begin with the word **program**.)

The put_string procedure is defined by nesting it within a program. Using our previous example, type in and run this program:

```
program hello;
{ say hello }

procedure put_string (s : string;
    x, y, f, p : integer);
 { display string s in font f, size p, at (x, y) }

begin { put_string }
 textfont(f);
 textsize(p);
```

```
 moveto(x, y);
 drawstring(s)
end;

begin { hello }
 put_string('Hello there, world!', 5, 100, 5, 14)
end.
```

Note that the arguments given in the procedure call match up with the argument list in the procedure heading.

In Pascal, all subprograms are nested within programs. In this book, subprogram nesting will usually only be indicated, rather than shown explicitly. However, it is important for you to type the subprogram text in the order indicated. For example, the current program would be written as

```
program hello;
{ say hello }

{ Insert procedure put_string }

begin { hello }
 put_string('Hello there, world!', 5, 100, 5, 14)
end.
```

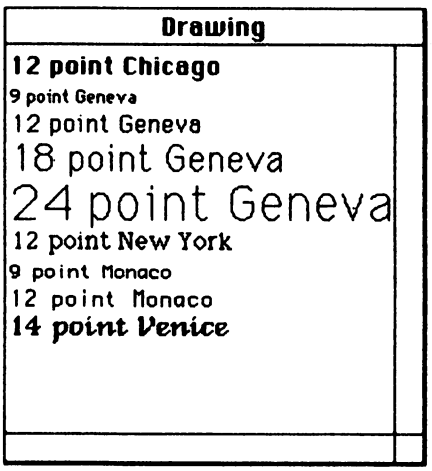
This indicates the text of the put_string procedure belongs at the position of the comment { *Insert procedure put_string* }. For the purposes of this text, a simple rule applies: whenever you see a comment { *Insert procedure x* } or { *Insert function y* }, type in the procedure or function itself.

A procedure may be called more than once within the same program. For example, a series of calls to put_string can be used to display all fonts normally available on the Macintosh Pascal disk:

```
program system_fonts;
{ display system fonts }

{ Insert procedure put_string }

begin { system_fonts }
 put_string('12 point Chicago', 2, 15, 0, 12);
 put_string('9 point Geneva', 2, 30, 1, 9);
 put_string('12 point Geneva', 2, 45, 1, 12);
 put_string('18 point Geneva', 2, 65, 1, 18);
```

```
  put_string('24 point Geneva', 2, 90, 1, 24);
  put_string('12 point New York', 2, 105, 2, 12);
  put_string('9 point Monaco', 2, 120, 4, 9);
  put_string('12 point Monaco', 2, 135, 4, 12);
  put_string('14 point Venice', 2, 150, 5, 14)
end.
```

The result:



Obviously, you could have written a program to do the same thing without using a procedure. But this would involve four procedure calls for *each* of the nine displayed strings, for a total of 36 lines in the program. This reveals one of the most important reasons for using subprograms: the code for a common operation involving more than a few statements may be written once as a subprogram. Whenever the operation needs to be carried out, a single procedure (or function) call can be used instead of the equivalent statements. This approach is much easier to program (and less prone to error) than writing out a series of statements. Depending on the situation, it can also save a considerable amount of memory.

A second, equally important motive for using procedures is that they help keep the size and complexity of your program under control. It is difficult for most programmers to keep track of all the details involved in large program segments: deep statement nesting and a multiplicity of variables can be extremely hard to figure out. Chopping out chunks of code for insertion into procedures can make the program easier to understand by breaking it into several smaller, rela-

tively independent pieces.

For example, let's return to the word-counting program of Chapter 7. (Review the program now if you wish.) You might decide to implement the following pieces of the program as separate procedures:

- Display the instructions
- Delete leading blanks from the string
- Split off the first word from a string
- Report the results.

If we were to write a procedure for each of these tasks, the resulting main program might look like this:

```
program wordcount;
{ count words in input text }

 var
  line, word : string;
  wordcount, sumlength : integer;

{ Insert procedure instruct }
{ Insert procedure delete_leading_blanks }
{ Insert procedure chop_first_word }
{ Insert procedure report }

begin { wordcount }
 instruct;
 sumlength := 0;
 wordcount := 0;
 readln(line);
 while length(line) > 0 do
  begin
   repeat
    delete_leading_blanks(line);
    if length(line) > 0 then
     begin
      chop_first_word(line, word);
      wordcount := wordcount + 1;
      sumlength := sumlength + length(word)
     end
   until length(line) <= 0;
   readln(line)
  end;
 report(wordcount, sumlength)
end.
```

Note that the four procedures (instruct, delete_leading_blanks, chop_first_word, and report) are inserted into the program just after the variable declarations. The main program is now much easier to take in at a glance and the mnemonic names for the procedures make it easier to follow what the program is doing.

The *instruct* procedure for counting words is easy to write:

```
procedure instruct;
 { display instructions for wordcount }

begin { instruct }
 write('This program counts the number of words');
 write(' in an arbitrary number of input lines and');
 writeln(' computes the average word length.');
 writeln;
 write('Please enter the input text.');
 write(' Press the Return key after each line.');
 writeln(' Enter an empty line when done.');
 writeln
end;
```

When a procedure has no arguments, the parentheses are omitted in the procedure header, just as they are in the procedure call.

The report procedure is also rather easy to write:

```
procedure report (wordcount, sumlength : integer);
{ report word-counting results }

var
 avelen : real;

begin { report }
 writeln('Number of words: ', wordcount : 1);
 if wordcount > 0 then
  begin
   avelen := sumlength / wordcount;
   writeln('Average word length: ', avelen : 1 : 1)
  end
end;
```

Note there is nothing wrong with using the same variable names in the procedure header and the main program.

A procedure may declare its own variables and named constants in declaration sections separate from the main program. Such identifiers are termed *local* identifiers; they are defined within a subprogram and can only be accessed while that subprogram is executing. Think of the variable avelen as coming into existence only when the report procedure is entered and winking out of existence once control is passed back to the main program. The variables declared in the main program, on the other hand, are termed *global* variables because they exist throughout the entire program's execution.

The delete_leading_blanks procedure is slightly more complex. The procedure call

```
delete_leading_blanks(line);
```

must change the **string** variable line by deleting its leading blanks; the procedure then uses the variable to return the result of the procedure. You'll remember from our discussion in the previous chapter that this implies that the argument to delete_leading_blanks must be a variable rather than a value. This fact must be noted in the procedure heading when delete_leading_blanks is written; variable arguments to subprograms are noted in a subprogram heading by prefacing the arguments with the reserved word **var**:

```
procedure delete_leading_blanks (var s : string );
 { delete leading blanks from s }

 const
  BLANK = ' ';

begin { delete_leading_blanks }
 while pos(BLANK, s) = 1 do
  delete(s, 1, 1)
end;
```

The procedure chop_first_word has two arguments, both variables:

```
procedure chop_first_word (var line, word : string );
 { extract first word in line into word, delete from line }

 const
  BLANK = ' ';
```

```
var
 nextblank : integer;

begin { chop_first_word }
 nextblank := pos(BLANK, line);
 if nextblank = 0 then
  begin
   word := line;
   line := ''
  end
 else
  begin
   word := copy(line, 1, nextblank - 1);
   delete(line, 1, nextblank)
  end
end;
```

We have now specified all the pieces of this version of the word-counting program. Type it in and verify that it works the same way as the previous version. (You may find it easier to cut and paste the previous version, if you saved it, than to type an entire new program.)

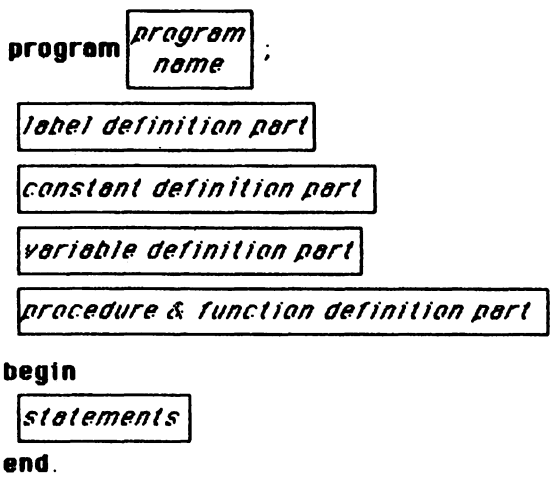


Figure 9-2.

Pascal program syntax (still incomplete)

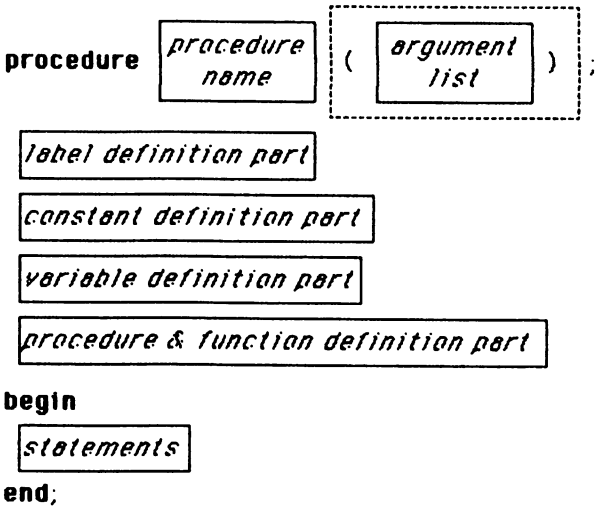


Figure 9-3.

Procedure definition syntax

Procedures (and functions) called from the main program must be defined between the variable-declaration section and the first **begin** of the main program. This adds a new detail to the sketch for Pascal program syntax, as shown in Figure 9-2.

Since procedures are "little programs" their syntax matches that of programs very closely. Figure 9-3 shows an incomplete sketch of procedure syntax.

As mentioned previously, the variables declared within a procedure are local to that procedure. Similarly, any labels, constants, procedures, and functions are also local; they may be referenced only while the procedure is executing. Note especially that there is no arbitrary restriction on the amount of nesting of procedures and functions. You may declare procedures within procedures that are themselves defined within procedures, and so on.

The procedure heading's argument list is used if and only if the procedure expects arguments. It is broken up into one or more argument sections separated by semicolons. Each argument section declares one or more arguments of a single type. Pascal is flexible as to how you write the argument list; if you wish, you may declare each argument in a separate

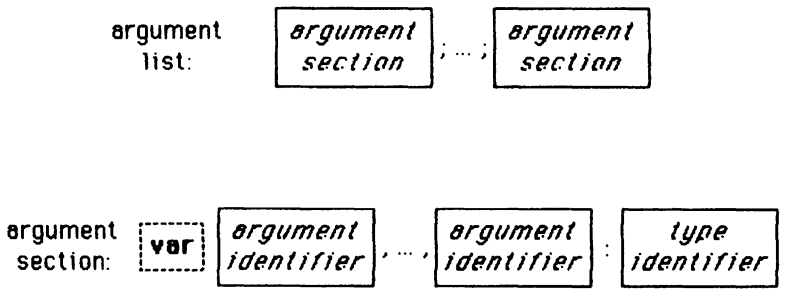


Figure 9-4.

Argument list syntax

section, or you may group arguments of the same type into sections. However, all arguments in one section must be either variable or value parameters, not a mixture of both. Each argument section is automatically placed on a new line and contains the following:

- The reserved word **var** if the arguments in the section are variable arguments.
- A list of one or more argument identifiers separated by commas.
- A colon followed by the type of arguments named in the section.

All this syntax is summarized in Figure 9-4.

It is important to appreciate the difference between variable and value arguments. For example, consider the following procedure called with two integer-value arguments:

```
procedure swapvals (x, y : integer);
 { swap values of x and y }

 var
  t : integer;

begin { swapvals }
 t := x;
 x := y;
 y := t;
 writeln('x = ', x : 2, ' y = ', y : 2)
end;
```

What would you guess the output would be if the *swapvals* procedure were called with this program?

```
program val_test;
{ test value arguments }

var
 x, y : integer;

{ Insert procedure swapvals }

begin { val_test }
 x := 31;
 y := 6;
 writeln('x = ', x : 2, ' y = ', y : 2);
 swapvals(x, y);
 writeln('x = ', x : 2, ' y = ', y : 2)
end.
```

Guess first, then try it. Were you surprised? The results you obtain should appear as follows:

```
x = 31 y =  6
x =  6 y = 31
x = 31 y =  6
```

The values of x and y switch places inside the procedure; this is shown by the second output line. But somehow the values get set back to their original values when control passes back to the main program. What happened?

The problem lies more in our misleading variable naming than anything else. The arguments passed to the swapvals procedure are the values of x and y from the main program, not the variables themselves. The variables x and y in the procedure are temporary (local) values that exist only while the procedure is executing. The procedure call initializes the local variables x and y to the values of the global variables x and y, but the local variables occupy a different region of memory from that of global variables. Changing the values of the local variables does not change the values of the global variables.

Now consider a procedure similar to swapvals, this time using variable arguments:

```
procedure swapvars (var u, v : integer);
 { swap variables u and v }

 var
  t : integer;

begin { swapvars }
 t := u;
 u := v;
 v := t;
 writeln('u = ', u : 2, ' v = ', v : 2)
end;
```

You can test this procedure with a slight modification to the previous program:

```
program var_test;
{ test variable arguments }

 var
  x, y : integer;

{ Insert procedure swapvars }

begin { var_test }
 x := 31;
 y := 6;
 writeln('x = ', x : 2, ' y = ', y : 2);
 swapvars(x, y);
 writeln('x = ', x : 2, ' y = ', y : 2)
end.
```

Again, predict the output before you run the program. The results should look like this:

```
x = 31 y =  6
u =  6 v = 31
x =  6 y = 31
```

This time changing the values of the local arguments inside the procedure *does* change the values of the variables passed to the procedure, because these arguments were specified as variables. A variable argument used inside a procedure, then, refers to the same region of the computer's memory as the actual variable passed to the procedure. (This is true even

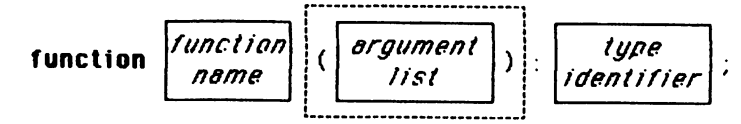


Figure 9-5.

Function header syntax

when, as here, the arguments in the procedure and the variable have different names.)

There is a simple rule to follow in deciding whether to declare arguments as values or variables. When you want subprogram arguments to return results, you must specify them as variable arguments. If you do *not* need the arguments to return results, you should specify value arguments instead.

WRITING FUNCTIONS

Nearly everything said in the previous section concerning procedures also applies to functions; there are only a few additional rules to follow when writing functions.

First, the form of the function heading is slightly different. Remember that a function call represents a value. In addition to specifying the name of the function and the arguments passed to the function, you must also specify the type of value represented by the function. Function heading syntax is shown in Figure 9-5.

As our first example, let's write a function called days_in_month that returns the number of days in a given month in a given year. For example, the line:

```
writeln(days_in_month(2, 1985))
```

should produce the output 28, the number of days in February, 1985.

We produced the necessary code for this function in our discussion of the **case** statement in Chapter 4. Adapting the code to a function is straightforward:

```
function days_in_month (mo, yr : integer) : integer;
 { return number of days in month of specified year }

begin { days_in_month }
 case mo of
  9, 4, 6, 11 :
   days_in_month := 30;
  1, 3, 5, 7, 8, 10, 12 :
   days_in_month := 31;
  2 :
   if is_leap(yr) then
    days_in_month := 29
   else
    days_in_month := 28;
  otherwise
   days_in_month := 0
 end
end;
```

There are a couple of notable features here. First consider the lines assigning values to the variable days_in_month, for example:

```
days_in_month := 30;
```

Assignment statements such as these give the function its value each time it is executed. There must be at least one assignment to the function's name in the function, and such an assignment must always be made at least once whenever a function is executed; otherwise, the value returned by the function will be undefined.

You may have already noticed that days_in_month calls another function, is_leap, in the case where the month considered is February. Although we haven't done so as yet, there is no reason why one of your own functions can't call another one of your functions. In general, any subprogram can call any other subprogram. (You need to be a little careful how the subprograms are ordered, though; we'll consider the rules in the next section.)

The call to is_leap appears between an **if** and a **then**, which means it must represent a Boolean value; if it didn't, a type-mixing bug box would appear. Examining the code reveals is_leap must return TRUE if the year is a leap year and FALSE if it isn't. (This is also suggested by the function's name.) The function, then, might be written this way:

```
function is_leap (yr : integer) : Boolean;
{ return TRUE if year is a leap year, else FALSE }

begin { is_leap }
 is_leap := (yr mod 4 = 0) and (yr mod 100 <> 0)
                                    or (yr mod 400 = 0)
end;
```

You may wish to compare these two functions with the original program in Chapter 4. Here is a test main program to verify that the functions work as expected:

```
program day_test;
{ tell number of days in month }

 var
  month, year : integer;

{ Insert function is_leap }
{ Insert function days_in_month }

begin { day_test }
 write('Enter the month number:');
 readln(month);
 write('Enter the year:');
 readln(year);
 write('There are ', days_in_month(month, year) : 1);
 writeln(' days in month ', month : 1, ' of ', year : 1)
end.
```

Functions need not be complex, as we've seen. Often, you'll want to write functions of your own design to extend the library functions. For example, this function accepts two arguments, lo and hi, and returns a random integer value that's between lo and hi inclusive:

```
function randint (lo, hi : integer) : integer;
{ return random integer between lo and hi, inclusive }

begin { randint }
 randint := lo + random mod (hi - lo + 1)
end;
```

As another example, the following function accepts a character value as an argument; if the character is an upper-

case letter, the function returns the equivalent lowercase letter. (If the argument is not an uppercase letter, the argument's value is returned unchanged.)

```
function to_lower (ch : char) : char;
{ convert ch to lowercase }

begin { to_lower }
 if (ch >= 'A') and (ch <= 'Z') then
  to_lower := chr(ord(ch) - ord('A') + ord('a'))
 else { don't change any but uppercase }
  to_lower := ch
end;
```

We will present a program that uses both the randint and to_lower functions in a while; for now, you might try to write simple programs that test them as we did for days_in_month.

As a final example, let's return to the golden program we developed and debugged in Chapter 5; it used the method of bisection to find the root of the equation

$$x^2 + x - 1 = 0$$

The golden program required calculating the value of the expression on the left side of this equation for different values of x. This time let's write a function to carry out the calculation:

```
function f (x : real) : real;

begin { f }
 f := sqr(x) + x - 1.0
end;
```

Functions are commonly used to isolate such calculations, keeping them separate from the logical flow of the program itself. For example, we might write another function that determines whether two values have opposite signs:

```
function oppsign (u, v : real) : Boolean;
{ return TRUE if u & v are of opposite sign, else FALSE }

begin { oppsign }
 oppsign := ((u < 0.0) and (v > 0.0)) or ((u > 0.0) and (v < 0.0))
end;
```

A version of the golden program that uses these functions might look like this:

```
program golden;
{ find the golden ratio! }

 const
  EPSILON = 1.0e-6;

 var
  xmid, x1, x2 : real;

{ Insert function f }
{ Insert function oppsign }

begin { golden }
 write('Enter lower bound for solution: ');
 readln(x1);
 write('Enter upper bound for solution:');
 readln(x2);
 repeat
  xmid := (x1 + x2) / 2.0;
  if oppsign(f(x1), f(xmid)) then
   x2 := xmid
  else
   x1 := xmid
 until abs(f(xmid)) < EPSILON;
 writeln('Solution is ', xmid : 16)
end.
```

This version of golden is considerably cleaner than the previous one, mainly because it relies on functions to do the nuts-and-bolts calculations and confines itself to the implementation of the bisection algorithm. Note also that it is much easier to modify this version of the program to find the roots of different equations; all you need to do is to change the function f.

SCOPE AND NESTING

Pascal has a simple rule that governs placement of procedures and functions in your programs:

Procedures and functions must be defined before they are used

In attempting to make sense out of your program, Macintosh Pascal scans through it from beginning to end. When it encounters a call to a procedure or function not yet defined in your program, it considers it to be an error, even if the procedure or function actually gets defined later in the program.

As an example, re-examine the test_days program in the previous section that used the functions days_in_month and is_leap to calculate the number of days in a month. The function is_leap must precede days_in_month, because it is called in days_in_month. If, instead, is_leap appeared in the program someplace after days_in_month, a not-yet-defined bug box would appear, and a "thumbs-down" would appear on the line with the call to is_leap within days_in_month.

Things become complicated only slightly when your program contains procedures and functions nested within other procedures and functions. Remember that identifiers defined within a subprogram are local to that subprogram and can't be used outside the subprogram. This rule applies to all identifiers: named constants, variables, as well as procedure and function names. So, in order for a subprogram call to be legal, the called subprogram's definition must not only precede the call, but it must also be "visible" to the caller; it can't be buried within another subprogram.

Pascal has a few rules, called *scope rules*, that govern when an identifier defined in one part of the program may be used in another part:

- An identifier defined in a subprogram (or the program itself) may be used later within that same subprogram (or program).
- An identifer defined in a subprogram (or the program itself) may be used in any later subprogram nested within that subprogram (or program).
- If a subprogram can see two identifiers with the same name, the more local definition takes precedence.

This final rule allows subprograms to be written independently of the programs that call them. You need not worry that your use of a name within a subprogram will conflict with the use of the same name in another part of your pro-

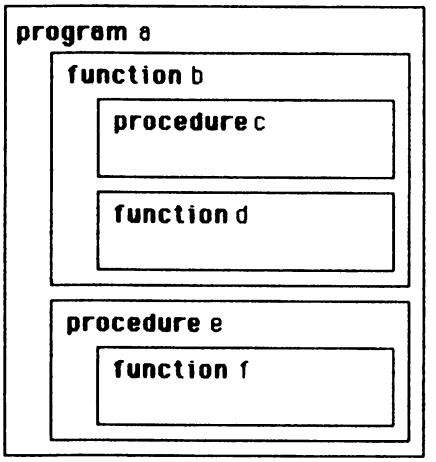


Figure 9-6.

Nested procedures and functions

gram. An identifier defined within a subprogram will always retain that definition within the subprogram, no matter how the identifier is defined outside the subprogram.

Pascal's scope rules may seem simple, even trivial; as usual, however, they can offer some surprises to the novice. To make sure you know the implications of the rules, consider the schematic outline of a program in Figure 9-6, which shows procedure c and function d nested within function b, and function f nested within procedure e.

The scope rules imply, for example, that identifiers defined in the program itself are accessible to all later parts of the program, including all nested subprograms. Identifiers defined within procedure c, on the other hand, are only visible within procedure c. How the scope rules apply to the schematic program are summarized in Table 9-1.

Which subprograms can call which? This question can be answered by observing that a subprogram is defined in the program or subprogram that encloses it. Example: procedure c is defined within function b, so procedure c is callable anywhere identifiers defined within function b are accessible: function b itself, procedure c, and function d. A similar argument can be carried through for all subprograms; a summary for our example program is shown in Table 9-2.

Table 9-1.

Scope Rules for Figure 9-6

Identifiers Defined In:	Are Accessible In:	Aren't Accessible In:
program a	a, b, c, d, e, f	-none-
function b	b, c, d	a, e, f
procedure c	c	a, b, d, e, f
function d	d	a, b, c, e, f
procedure e	e, f	a, b, c, d
function f	f	a, b, c, d, e

Table 9-2.

Allowable Subprograms for Figure 9-6

Statements in	Can Call Subprograms	Can't Call Subprograms
program a	b, e	c, d, f
function b	b, c, d	e, f
procedure c	b, c	d, e, f
function d	b, c, d	e, f
procedure e	b, e, f	c, d
function f	b, e, f	c, d

PROGRAMMING WITH PROCEDURES AND FUNCTIONS

There is nothing like experience in learning to program, and the proper use of procedures and functions is worth learning well. In this section we will develop our most ambitious program so far, a program to play the dice game known as craps.

Craps involves an indefinite number of rounds. At the start of each round the player places a bet; a loss subtracts the amount of the bet from the player's funds and a win adds the amount of the bet to the player's funds. The game ends either when the player quits voluntarily or is no longer able to come up with the "house minimum" bet.

This description is all we need to design the main program. A pseudo-code summary of the playing algorithm might go as follows:

```
initialize funds
repeat
    show player's current balance
    play a round
    if funds are low
        toss player out
    else
        ask if player wants to continue
until player quits, voluntarily or otherwise
```

This pseudo-code is detailed enough to translate directly into Pascal:

```
program craps;
{ Plays simple craps game }

 const
  INITIAL_STAKE = 1000.00;
  HOUSE_MIN = 1.00;

 var
  funds : real;
  done : Boolean;

{ Insert function get_yes_or_no }
{ Insert procedure playround }

begin { craps }
 funds := INITIAL_STAKE;
 repeat
  page;
  writeln('Your current balance: $', funds : 1 : 2);
  playround(funds);
  if funds <= HOUSE_MIN then
   begin
    writeln('You're busted!');
    done := TRUE
   end
  else
   done := (get_yes_or_no('Play again?') = 'n')
 until done;
 writeln;
 writeln('You are left with $', funds : 1 : 2)
end.
```

It is important to note that we have first concentrated on the "outer" level of the game; consideration of the detailed rules of the game is delayed until later. Many beginning programmers make the mistake of worrying about too much, too soon in the program design process. All we are concerned about here is getting the simplest part of the logic nailed down: keeping track of the money and deciding whether to play another round.

The main program calls one procedure and one function. The procedure playround will contain all the details on playing one round of the game. The get_yes_or_no function, on the other hand, does something much simpler. It gets the answer to a yes-or-no question from the player. Let's do that one first.

On the Macintosh, there are two ways to get input from the user of a program: the keyboard or the mouse. Using the mouse to get a yes-or-no response would probably involve something like a dialog box, and that is slightly beyond our current capabilities. The keyboard is our choice for now.

We want our get_yes_or_no function to be relatively forgiving of input errors on the part of the player. Although we can't do anything about the case where a player means to answer yes but types no instead, we can ignore anything but a yes or no response; this allows the most common typing mistakes to be ignored. Let's consider a yes response to be a press of the Y key on the keyboard, and a no response to be a press of the N key. (This should work whether the SHIFT key is pressed at the same time or not; Y also means yes.) The following code gives us the response we want:

```
function get_yes_or_no (prompt : string ) : char;
 { get 'y' or 'n' response }

 var
  ch : char;

 { Insert function to_lower }

 begin { get_yes_or_no }
  write(prompt, '(y/n): ');
  repeat
   read(ch);
   ch := to_lower(ch);
   if (ch <> 'y') and (ch <> 'n') then
    begin
```

```
    sysbeep(1);
    write(chr(8), ' ', chr(8))
   end
  until (ch = 'y') or (ch = 'n');
  get_yes_or_no := ch
 end;
```

Note the use of chr(8) in the part of the function that handles invalid input. chr(8) represents the backspace character. When the statement

```
write(chr(8), ' ', chr(8))
```

is executed, the Text window cursor backs up one position, writes a space to erase whatever erroneous character was typed in, and backs up again to the same position.

Testing what we've got so far is possible by using a temporary "stub" version of the playround procedure:

```
procedure playround(var funds : real);
{ stub procedure playround }

begin { playround }
 funds := funds / 2.0
end;
```

Try typing in the entire program with this stub procedure and run it; verify that you may exit from the program voluntarily at any time or wait until your funds dwindle away and you are forced out. Also try typing erroneous characters in response to the "Play again?" prompt.

Once you are convinced that the program works as it's supposed to, we can continue with the design. We now need to write the part of the program that plays a single round of craps. The rules we'll use are easy to state:

- Each round consists of one or more throws of a pair of dice.
- A throw of 2, 3, or 12 on the initial throw is an immediate loss.
- A throw of 7 or 11 on the first throw wins.
- Any other number rolled on the first throw becomes the player's point. Throws continue until either that point occurs again (in which case the player wins) or a 7 is rolled (in which case the player loses).

You may wish to come up with your own playround procedure at this point. If so, try to start with a rough pseudo-code version at first, refining it later if necessary. Don't be reluctant to invoke calls to subprograms within your design if you feel that the procedure is getting too complex. (You can always worry about writing the new subprograms later.)

Here is one possible pseudo-code for the playround procedure:

```
get bet for this round
repeat
    throw dice
    if first throw
        if player rolled a 2, 3, or 12
            player loses
        else if player rolled 7 or 11
            player wins
        else
            point <- roll
    else { second or later roll }
        if player rolled point then
            player wins
        else if player rolled a 7
            player loses
until player wins or loses
if player won
    add bet to funds
else
    subtract bet from funds
```

This pseudo-code is a straightforward description of the rules described: play continues until the player wins or loses, and the first roll is handled differently from subsequent rolls. Translation into Pascal is helped by delegating to subprograms the responsibilities for rolling the dice and setting the amount of the bet:

```
procedure playround (var funds : real);
 { play one round }

 var
  bet : real;
  thrownum, point, sum : integer;
  win, lose : Boolean;
```

```
{ Insert function get_bet }
{ Insert procedure waitclick }
{ Insert function throwdice }

begin { playround }
 bet := get_bet(HOUSE_MIN, funds);
 thrownum := 0;
 win := FALSE;
 lose := FALSE;
 repeat
  writeln('Click button to throw dice...');
  waitclick;
  sum := throwdice;
  writeln('You rolled a ', sum : 1);
  thrownum := thrownum + 1;
  if (thrownum = 1) then
   if (sum = 2) or (sum = 3) or (sum = 12) then
    lose := TRUE
   else if (sum = 7) or (sum = 11) then
    win := TRUE
   else
    begin
     point := sum;
     writeln('Your point is ', point : 1)
    end
  else if sum = point then
   win := TRUE
  else if sum = 7 then
   lose := TRUE
 until win or lose;
 if win then
  begin
   writeln('You won!');
   funds := funds + bet
  end
 else
  begin
   writeln('Sorry, you lose...');
   funds := funds - bet
  end
end;
```

The get_bet function is mainly concerned with checking that the player enters a bet that is at least the house min-

imum, but not more than the amount of money the player currently has:

```
function get_bet (min, max : real) : real;
{ get bet between min and max }

 var
  x : real;

begin { get_bet }
 repeat
  write('Enter your bet:');
  readln(x);
  x := round(100 * x) / 100;
  if (x < min) or (x > max) then
   writeln('Please enter an amount between $',
                              min : 1 : 2, ' and $', max : 1 : 2)
 until (x >= min) and (x <= max);
 get_bet := x
end;
```

For authenticity, the player should be required to do something to make the dice roll. In this program, the player presses and releases the Mouse button to make each roll. This procedure waits until the Mouse button has been pressed and released:

```
procedure waitclick;
{ Wait for mouse press and release }

begin { waitclick }
 while not button do
  ;
 while button do
end;
```

The only thing left to write is throwdice, the routine that "throws the dice" and returns their sum. A throw of a single die can be simulated by a random integer in the range 1 to 6, obtained from a call to the randint function written earlier. Two such calls to randint give the values for two dice. Adding these gives us the needed return value.

In addition to simply returning the values of the dice

rolled, let's also put into throwdice a provision to draw the dice in the Drawing window:

```
function throwdice : integer;
{ throw dice, return sum }

 var
  die1, die2 : integer;

{ Insert function randint }
{ Insert procedure drawdice }

 begin { throwdice }
  die1 := randint(1, 6);
  die2 := randint(1, 6);
  drawdice(die1, die2);
  throwdice := die1 + die2
 end;
```

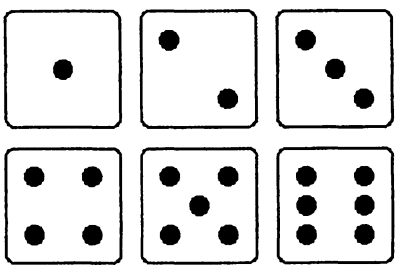
This is a natural point to test what we've written so far. Type in the new parts of the program, writing a stub procedure for drawdice. Run the program and verify that everything (except dice drawing) works as expected. This process of incremental design, testing, and (if necessary) debugging can be a real help in designing larger programs; it gets at least a part of the program working quickly, and can help reveal design flaws and dead ends while it's still relatively easy to recover from them.

Although dice drawing isn't too difficult, we will do our best to make it as easy as possible. First, we'll write drawdice as simply as possible: we'll make two calls to a procedure that draws a single die at a specified position in the Drawing window:

```
procedure drawdice (d1, d2 : integer);
 { draw dice }

 const
  DIE_1_X = 20;
  DIE_1_Y = 70;
  DIE_2_X = 120;
  DIE_2_Y = 70;
```

```
{ Insert procedure drawdie }

begin { drawdice }
 drawdie(d1, DIE_1_X, DIE_1_Y);
 drawdie(d2, DIE_2_X, DIE_2_Y)
end;
```

All that's left, then, is the drawdie procedure. We'll be content to draw a simple two-dimensional picture of a die face. Most dice have slightly rounded corners, so the outline of the die face might best be drawn with calls to rounded-rectangle routines. The spots can be drawn using calls to paintcircle. What we need is a simple method to decide where on the face to draw the dots, given the die's value. We have six possibilities:



It's often helpful to turn around a problem slightly to see if it has an easier solution. In this case, instead of asking "For a given die value, where are the spots painted?" let's ask instead "Consider a given spot; for what die values is that spot painted?" Examining the possible spot arrangements gives the following observations:

- The center spot is needed only if the die is odd: 1, 3, or 5.
- The spots in the northwest and southeast corners of the die face are displayed whenever the die value is greater than 1.
- The spots in the northeast and southwest corners are displayed if the die value is greater than 3.
- The east and west spots are displayed only if a 6 is rolled.

Putting all this into the drawdie procedure is primarily a matter of adjusting the relative sizes and positions of the die face and spots to get a reasonably accurate picture. Here is one solution:

```
procedure drawdie (d, x, y : integer);
{ draw die with value d at (x, y) }

  const
   DIESIZE = 60;
   DOTPOS1 = 15;  { DIESIZE div 4 }
   DOTPOS2 = 30;  { 2 * DOTPOS1 }
   DOTPOS3 = 45;  { 3 * DOTPOS1 }
   DOTSIZE = 5;
   R_OVAL_HT = 10;
   R_OVAL_WID = 10;

  begin { drawdie }
   eraseroundrect(y, x, y + DIESIZE, x + DIESIZE,
                          R_OVAL_HT, R_OVAL_WID);
   frameroundrect(y, x, y + DIESIZE, x + DIESIZE,
                          R_OVAL_HT, R_OVAL_WID);
   if odd(d) then
    paintcircle(x + DOTPOS2, y + DOTPOS2, DOTSIZE);
   if d > 1 then
    begin
     paintcircle(x + DOTPOS1, y + DOTPOS1, DOTSIZE);
     paintcircle(x + DOTPOS3, y + DOTPOS3, DOTSIZE);
     if (d > 3) then
      begin
       paintcircle(x + DOTPOS3, y + DOTPOS1, DOTSIZE);
       paintcircle(x + DOTPOS1, y + DOTPOS3, DOTSIZE);
       if d = 6 then
        begin
         paintcircle(x + DOTPOS1, y + DOTPOS2, DOTSIZE);
         paintcircle(x + DOTPOS3, y + DOTPOS2, DOTSIZE)
        end
      end
    end
  end;
```

You may want to experiment with the numbers in this procedure to see how the die appearance is affected.

This finishes our development of the game; try typing in the remainder of the program and try it out.

In developing this program, we have demonstrated yet another benefit of procedures and functions: using them allows program design to proceed in a logical, stepwise manner. Although the final program is much larger than anything we've written previously, no single part of the pro-

gram was particularly difficult to write or hard to understand. Working mostly with subprograms allowed us to concentrate on small pieces of the problem at any given time, rather than trying to tackle the entire program all at once. This is a lesson that will prove valuable in later chapters.

YOUR OWN DATA TYPES

10

Choosing a better data structure is an art, which we cannot teach.

B. KERNIGHAN AND P.J. PLAUGER
The Elements of Programming Style
(McGraw-Hill, 1978)

You can think of programs as consisting of two parts: an active part and a passive part. So far we have been concentrating on the active part of programs: the statements, procedures, and functions that make the program do something. In the background has been the passive, or acted-upon, part of programs: the information or *data structures* that the program manipulates to achieve the desired results. So far our data structures have been, simply, variables of the various types already built into Macintosh Pascal.

Just as a large program can be constructed out of smaller parts, however, data structures used in a program may be built up out of the pieces we have already considered. In addition, just as we can write procedures and functions to add to the subprograms available in the library, we can also create data types of our own to supplement the built-in data types.

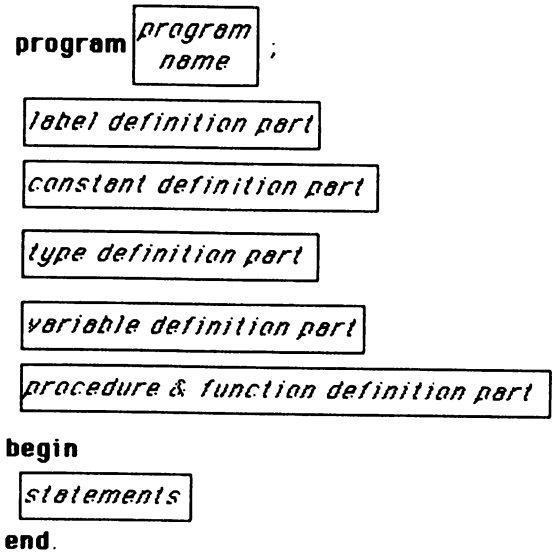


Figure 10-1.

Pascal program syntax (still incomplete)

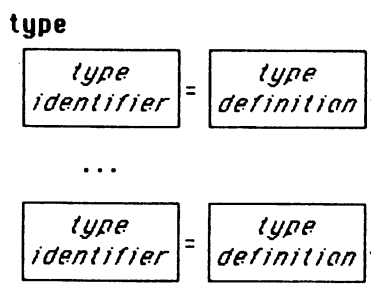


Figure 10-2.

Type definition part syntax

THE TYPE DEFINITION PART AND ENUMERATED TYPES

New data types of your invention are defined in the *type definition part* of programs and subprograms. The type definition part appears after the constant definition section and before the variable definition section. The way this affects the program syntax structure is shown in Figure 10-1. The syntax of the type definition part is sketched in Figure 10-2.

When you want to define your own data type, Pascal requires that you give it a name and follow that name with the actual definition. The name can be any legal identifier that isn't already being used in your program for some other purpose. The definition of the type can follow a number of different rules. Some of the simpler forms are discussed in this chapter.

The easiest data type definition is simply to define a synonym for an existing type. For example, consider the following type definition part:

```
type
 counter = integer;
 data_type = real;
 name = string[25];
```

This defines the new type counter, data_type, and name. Note that you are defining new variable types, not new variables. From this point on, the new type names can be used in the same way as the built-in types we've already discussed. You can now declare variables using these new types.

```
var
 count : counter;
 x : data_type;
 first, last, middle : name;
```

These variables can be used just as you would use variables of the existing synonymous type. You may use new type names in subprogram headings, for example:

```
function count_caps (n : name) : counter;
{ count uppercase letters in name n }

  ...
```

What is the purpose of defining synonyms for existing types? Why not just use the original names instead? There are a couple of reasons, one based on design considerations, the other on syntax considerations.

For reasons of design, defining synonyms for types increases the flexibility of the program and makes subsequent modification easier. For example, suppose you made the following definition of the distance type:

```
type
 distance = real;
```

Such a definition might be used to emphasize that some of the data used by your program represents actual, real-world distances. For example, the declaration

```
var
 d : distance;
```

is a convenient way to communicate to someone reading your program that the variable d will be storing a distance. And if you decide the distance values used in your program can accommodate more precision, most of the work involved is simply changing the type definition, for example:

```
type
 distance = double;
```

It is much easier to simply change a type definition at the beginning of your program than it is to search through the program and change only those occurrences of real variables representing distance to double variables.

The second reason to define synonyms of existing types is based in syntax. Pascal requires the types used in argument lists to procedure and function headings to be single identifiers. For example, the following function heading is illegal:

```
function count_caps (n : string[25]) : counter;
{ count uppercase letters in name n }

  ...
```

The correct way to define an argument of type **string**[25] is to use a synonymous type identifier instead, as we showed previously.

Type names defined in a type definition section follow the same scope rules as those explored in the previous chapter for constants, variables, and subprogram names. Types defined in the outermost shell of the program are visible to all later parts of the program, unless the type name gets another definition within a subprogram; in that case the new definition applies within that subprogram.

In addition to defining new names for types we already know about, you may also define completely new data types within the program. There are a number of type-definition methods available in Pascal; here we'll explore *enumerated* types.

An *enumerated* type is a collection of values to which you may assign names of your own choosing. As the name implies, an enumerated type is simply defined by listing all possible values that variables of the type may assume. Enumerated types are primarily useful when you want to represent a quantity that may take on a small number of possible values, such as the days of the week, the months of the year, and so on. (Other languages usually encode such values as small integers; this is considerably less reliable and readable than defining an enumerated type.)

Each value in the definition list must be represented by a legal Pascal identifier and the list must be enclosed by parentheses. Study these examples of enumerated type definitions:

```
type
 month_type = (JANUARY, FEBRUARY, MARCH, APRIL, MAY,
               JUNE, JULY, AUGUST, SEPTEMBER,
               OCTOBER, NOVEMBER, DECEMBER);
 computer_type = (APPLE, IBM_PC, MACINTOSH, VAX,
                  DEC10, COMMODORE_64);
 color_type = (RED, GREEN, BLUE, ORANGE, WHITE);
 day_type = (SUNDAY, MONDAY, TUESDAY, WEDNESDAY,
             THURSDAY, FRIDAY, SATURDAY);
```

Note: Each enumerated type definition is automatically formatted to take up a single line in a Macintosh Pascal program. Typographical limitations prevent us from showing some of these definitions as they would actually be formatted in your program.

These are only examples; it should be clear that you may define an enumerated type as a list of whatever named values

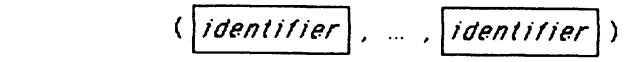


Figure 10-3.

Enumerated type definition syntax

you want. Follow the simple syntax shown in Figure 10-3. The enumerated type values have been written in uppercase to emphasize that they represent constant values, not variables. Since Macintosh Pascal ignores the case of identifiers (with one exception, noted later), you should feel free to adopt another style if you find it more attractive or useful.

Once you have defined an enumerated type, you're ready to define variables of that type:

```
var
  my_computer, your_computer : computer_type;
  chair_color, desk_color : color_type;
  this_month, last_month : month_type;
  yesterday, today, tomorrow : day_type;
```

Once defined, enumerated variables act much like other variables. You can assign them any legal value of their types, for example:

```
my_computer := MACINTOSH;
your_computer := IBM_PC;
desk_color := ORANGE;
chair_color := desk_color;
today := SUNDAY;
tomorrow := MONDAY
```

Pascal's strictures on type mixing also apply to enumerated types. All the following statements are invalid because they assign a value of one type to a variable of another:

```
last_month := APPLE;    { all illegal assignments }
yesterday := BLUE;
my_computer := today
```

The major restriction on the names you use for enumer-

ated type values is that they not conflict with other uses for that same name. For example, given the previous definitions, it would be a mistake to define fruit_type as

```
type
 ...
 fruit_type = (ORANGE, APPLE, BANANA, KIWI, MANGO);
```

because the identifier APPLE was previously defined as a value of computer_type and ORANGE was defined as a color_type value.

Enumerated types can be used to make programs more clear. For example, the days_in_month function written in Chapter 9 can be made more transparent by using the month_type enumerated type instead of an integer month number:

```
function days_in_month (mo : month_type;
      yr : integer) : integer;
 { return number of days in month of specified year }

begin { days_in_month }
 case mo of
  SEPTEMBER, APRIL, JUNE, NOVEMBER :
   days_in_month := 30;
  JANUARY, MARCH, MAY, JULY,
  AUGUST, OCTOBER, DECEMBER :
   days_in_month := 31;
  FEBRUARY :
   if is_leap(yr) then
    days_in_month := 29
   else
    days_in_month := 28
 end
end;
```

A call to this function used to determine the number of days in February, 1986, might appear as follows:

```
writeln(days_in_month(FEBRUARY, 1986) : 1)
```

In both cases, we have made it easier to see what the days_in_month function does and how it works by substituting the month names for the less-meaningful month numbers.

You might have noticed that it is acceptable to use a variable of an enumerated type as a **case** statement's controlling expression. In general, since any enumerated type you invent is considered to be an *ordinal* type, any of its variables can legally be used in a **case**'s expression.

Enumerated type variables can also be used as the loop control variable of a **for** statement. For example, the statements

```
sum := 0;
for mo := JANUARY to DECEMBER do
 sum := sum + days_in_month(mo, yr)
```

could be used to calculate the number of days in the year yr, assuming that the variable mo was declared as month_type. (Can you think of a more straightforward way to do the same thing using is_leap?)

Pascal's ordinal functions can be used with enumerated types as well. The pred and succ functions give the previous and next values specified in the type definition. These examples use the previous enumerated type functions:

```
pred(FEBRUARY)    is JANUARY
pred(MACINTOSH)   is IBM_PC
pred(BLUE)        is GREEN
pred(SATURDAY)    is FRIDAY
succ(JULY)        is AUGUST
succ(DEC10)       is COMMODORE_64
succ(RED)         is GREEN
succ(WEDNESDAY)   is THURSDAY
```

It is a mistake to apply the pred function to the first value in an enumerated type, or the succ function to the last value. For example, assume the variables this_month and next_month are declared as month_type. To set next_month to the month following this_month, if we write

```
next_month := succ(this_month)
```

it won't work if this_month is DECEMBER. Instead, we could write

```
if this_month = DECEMBER then
 next_month := JANUARY
else
 next_month := succ(this_month)
```

Notice that the relational operator = works to compare two enumerated type values. The other five relational operators can also be used since enumerated type values are ranked according to the order of their definition. An enumerated value is less than those following it in the defining list and greater than those that precede it.

The *ord* function applies to enumerated type values as well. It responds with the position in the list of its argument: the first value in the definition list has an ord value of 0, the second has the ord value 1, and so on. These examples are based on our previous definitions:

```
ord(JANUARY)     is 0
ord(JUNE)        is 5
ord(DECEMBER)    is 11
ord(MACINTOSH)   is 2
ord(MONDAY)      is 1
ord(SATURDAY)    is 6
```

Macintosh Pascal provides for input and output of enumerated type values. (This language feature is not present in Standard Pascal, so all our usual cautions about portability apply here.) For example, the day_test program from the previous chapter could be written as follows:

```
program day_test;
{ tell number of days in month }

type
 month_type = (JANUARY, FEBRUARY, MARCH, APRIL, MAY,
               JUNE, JULY, AUGUST, SEPTEMBER,
               OCTOBER, NOVEMBER, DECEMBER);

var
 month : month_type;
 year : integer;
```

```
{ Insert function is_leap }
{ Insert function days_in_month }

begin { day_test }
 write('Enter the name of the month: ');
 readln(month);
 write('Enter the year: ');
 readln(year);
 write('The number of days in ', month, ' ', year : 1, ' is ');
 writeln(days_in_month(month, year) : 1)
end.
```

The program now requests a month name instead of a number. The rule for entering enumerated values is that the entire name must be read in; no abbreviations or misspellings are allowed, but the case of the letters is ignored. Enter this program and try out different values for the month prompt.

When the program displays its results, an enumerated type gets printed just as it appears in the definition, including case. For example, with appropriate input, the day_test program might display this:

```
The number of days in FEBRUARY 1986 is 28
```

SUBRANGE TYPES

Another way you may define a type is as a *subrange* of an already-defined type. Let's start with some examples:

```
type
 die_value = 1..6;
 posint = 1..MAXINT;
 negint = -MAXINT..-1;
 digit = '0'..'9';
 uppercase = 'A'..'Z';
```

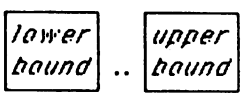


Figure 10-4.

Subrange type definition syntax

Defining a subrange type involves specifying a lower and upper bound on the possible values of the type and separating them with two periods; the syntax is shown in Figure 10-4. You should think of a subrange definition as making a sort of promise to Pascal: variables of this type will never get out of this range. For example, using the above definitions, vari-

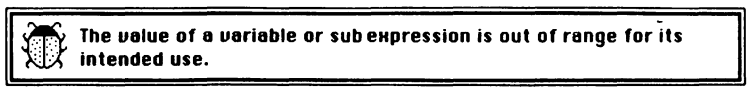
ables of the die_val type will only have six possible values: 1, 2, 3, 4, 5, and 6; anything else is now considered to be a violation of Pascal's rules.

Subranges may be of nearly any ordinal type, including enumerated types; in any case, the lower and upper bound values you specify must be of the same type, and the lower bound must be less than or equal to the upper bound. Note that reals are not an ordinal type, and you cannot make a subrange of reals. Both upper and lower bounds must be constant values; variables are forbidden. It is not legal to define an integer subrange in which either bound is out of the range −32767 to 32767.

Pascal behaves reasonably as long as your program doesn't break the promises it makes in subrange type definitions, but it extracts revenge if you slip up. For example, try the following program:

```
program subrange_lab;
{ test subrange type }

type
  legal_type = 1..10;

var
  num: legal_type;

begin { subrange_lab }
  write('Please enter a number between 1 and 10:');
  readln(num);
  writeln('You entered the number ', num: 1)
end.
```

Run this program twice, typing a legal value the first time (6, for example), and an illegal value (20, for example) the second time. The first time, everything proceeds normally, but entering the second value gives this bug box:



It is usually not a good idea to input a subrange value directly from the keyboard because it is too easy for the user of your program to type a value outside the subrange and

crash your program. Instead, accept values of the most general possible type, check their values, and then assign them to the subrange value desired. Using this strategy, the program would look like this:

```
program subrange_lab;
{ test subrange types }

type
 legal_type = 1..10;

var
 any_num : longint;
 num : legal_type;

begin { subrange_lab }
 write('Please enter a number between 1 and 10:');
 readln(any_num);
 if (any_num >= 1) and (any_num <= 10) then
  begin
   num := any_num;
   writeln('You entered the number ', num : 1)
  end
 else
  writeln('Sorry -- you entered the number ', any_num : 1)
end.
```

This version of the program is less easily crashed by accident.

Subrange variables are appropriate in larger programs where they act as guarantees to you and to anyone reading your program that no variable will take on unexpected values. As such they are an easy way to maintain confidence in the internal consistency of a program. Since they cause a program to crash whenever they get out of range, the logic errors involved in setting them to illegal values can be easily and quickly traced.

TYPE TAXONOMY

Before we go on to more complex data structures, it's worth stepping back a bit and summarizing the types we know

Table 10-1.

Type Classifications

Real Types	Ordinal Types	Integer Types	Simple Types	String Types
real	Boolean	integer	any ordinal type	**string**
double	char	longint	any real type	**string[ ]**
extended	any integer type	integer subranges		
computational	enumerated types			
	subrange types			

about so far. These are classified for us in Table 10-1.

The rules for handling different kinds of types are somewhat formal, and we don't need to discuss every last detail and nuance here. We will, however, discuss a few restrictions you'll need to know in writing your own programs.

First, let's reexamine the subrange_lab program from the previous section; note we assigned a value of a longint type (the variable any_num) to a variable of a legal_type (subrange-of-integer) type. Pascal allows this because the longint type and the legal_type type are *assignment-compatible.* The rules that determine whether a value of one type may be assigned to a variable of another type are lengthy, but worth examining:

- Any value of one type may be assigned to a variable of identical type.
- A value of any real type can be assigned to a variable of any other real type if the value is within the range of the target variable's type.
- An integer-type value may be assigned to any real-type variable.
- Any ordinal value may be assigned to a variable of a compatible ordinal type if the value is a legal value of that variable.
- A character value may be assigned to a string variable.
- A string variable of length 1 may be assigned to a character variable.
- A string value may be assigned to a string variable if the string value's length is less than or equal to the declared size of the string variable. (Remember the distinction between a string's length and its size.)

These rules are not yet complete, but they're adequate for now. Note that the second rule mentions "compatible" ordinal types. Two ordinal types are compatible if

- They are identical types.
- Both types are some kind of integer.
- One type is a subrange of another, or both are subranges of the same parent type.

Finally, we need to define what we mean by identical types. Although you might think this is too obvious to mention, two types are identical if they have the same type identifier (such as char or integer) or if one type was defined as a synonym of the other. When a subprogram expects a variable argument of a certain type, the variable specified in the argument list must be of an identical type, not just a compatible type.

Most of the rules we've just described we actually have been using informally all along. Generally, Pascal allows you to do the things that make sense (assigning a long integer value to an integer subrange variable, for example). On the other hand, Pascal frowns upon attempts to mix types beyond the rules here; assigning a character value directly to an integer variable will always be illegal in Pascal.

There is a shortcut method often used to declare variables of new types that bypasses the type-definition section. In Pascal you may define the type of a variable directly, when the variable is declared. Examples:

```
var
 finger : (THUMB, INDEX, MIDDLE, RING, LITTLE);
 card : 1..52;
 hour: 0..23;
```

Although legal, such shortcut definitions are often less flexible than the normal method of defining the new type in the type-definition section. Note that a type you define in the variable definition section has no name. And without a name, a variable with a shortcut type definition often can't be passed to a subprogram as an argument.

This concludes our look at simple, single-value data types. In the next chapter we'll delve into more complex data type constructions.

STRUCTURED DATA TYPES: ARRAYS, RECORDS, AND SETS 11

There hardly exist any programs of relevance outside the classroom which do not employ repetitions and arrays (or analogous data structures).

N. WIRTH
Programming in Modula-2
(Springer-Verlag, 1983)

So far we have limited our discussion to Pascal data types that store a single value at any given time. While useful, such *simple* data types are often too limited to accomplish more complex tasks. In this chapter we'll explore data types called *structured* types, which can be used to store many values at once. We will discuss three classes of structured types: *arrays*, *records*, and *sets*.

ARRAYS

The array is a data type common to many programming languages, including Pascal. You may think of an array as simply a collection of variables of the same type. Each individual variable is called an *element* of the array. To define an array type, Pascal requires you to specify the following information.

- The type of the array elements. (This is called the *base type* of the array.)
- The number of elements in the array.
- The method used to access each element of the array.

As usual, let's examine a concrete example first. Consider the type declaration

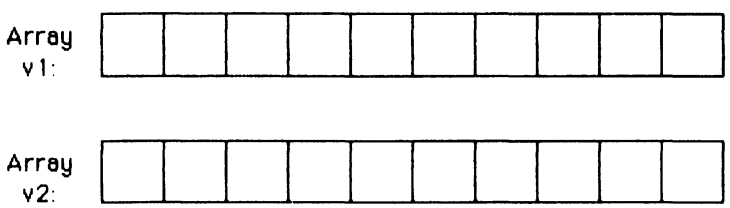
```
type
  vector_type = array[1..10] of real;
```

This type declaration tells Pascal that variables of type vector_type will contain real numbers (the base type). The subrange 1..10 inside square brackets in the definition implies that each vector_type variable will contain ten distinct real values. This also tells Pascal that vector_type array elements will be referenced using an integer between 1 and 10.

As you saw in the previous chapter, once a type has been defined you may declare variables of that type:

```
var
  v1, v2: vector_type;
```

This declares two variables, v1 and v2, both of type vector_type. Again, each array variable really should be thought of as containing a number of sub-variables; in this case, v1 and v2 each hold 10 real numbers as specified in the type declaration. We might picture this as follows:



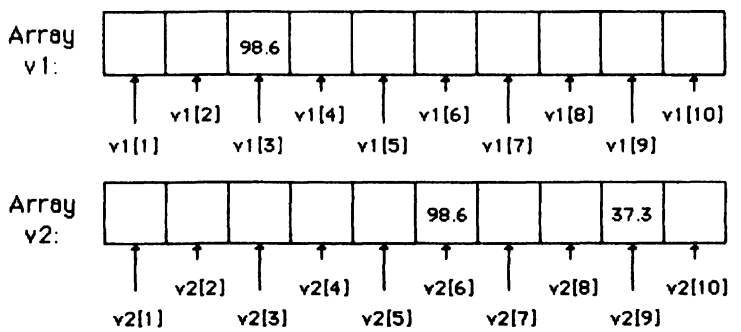
Here each large, rectangular box, symbolizing an entire array, is subdivided into ten smaller boxes, each large enough to hold a real value.

Programs access individual elements of an array by specifying the array name followed by a *subscript* in square brackets. The subscript follows the rules laid out in the type

definition. In our example, the subscript must be an integer value in the range of 1 to 10. The array name and subscript combination acts precisely as if it were a variable of the base type (real, in this case). That means you may assign values to array elements in the usual way, either by assigning them directly or reading values from the keyboard. For example:

```
v1[3] := 98.6;
v2[9] := 37.3;
v2[6] := v1[3]
```

The effect of these assignment statements is to place the value 98.6 into the third element of v1, the value 37.3 into the ninth element of v2, and then to copy the third element of v1 into the sixth element of v2. This might be pictured as follows:



The array name without a subscript is understood to refer to the array as a whole; that means to all ten real values contained in the array in our example. In many ways, the array name may be used just as any other variable name; for example, the assignment statement

```
v2 := v1
```

acts just as you might suspect it would: all ten real values held in the array v1 are copied into the corresponding elements of v2. In other words, the assignment is a very compact shorthand for these ten individual assignments:

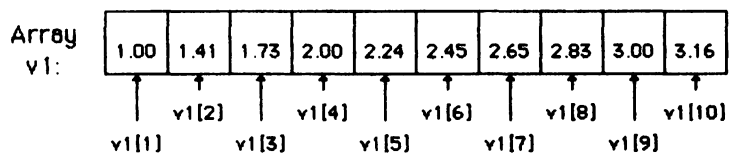
```
v2[1] := v1[1];
v2[2] := v1[2];
v2[3] := v1[3];
```

```
...
v2[9] := v1[9];
v2[10] := v1[10]
```

You are not restricted to using constants as subscripts. Pascal considers array subscripts to be expressions evaluated at the time the program is run. For example, this **for** loop initializes all elements of the array v1 to consecutive square roots:

```
for i := 1 to 10 do
  v1[i] := sqrt(i)
```

After the **for** is executed, all elements of v1 have defined values:

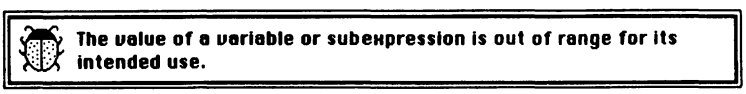


The only restrictions in constructing a subscript expression are that the expression be of the correct type and that its value fall into the range specified by the definition of the array type. Accidental violation of this latter restriction is the most common pitfall in writing programs that use arrays. It is all too easy to attempt to access an array element using a subscript that is out of bounds according to the original type definition. You may have guessed that Pascal doesn't allow such mistakes to pass without comment; to see what happens, try the following program that incorporates the definitions we've been using:

```
program array_lab;
{ Experiments with arrays }

type
 vector_type = array[1..10] of real;

var
 v1 : vector_type;
 i, which : integer;
```

```
begin { array_lab }
 for i := 1 to 10 do
  v1[i] := sqrt(i);
 while TRUE do
  begin
   write('Enter array index(1-10):');
   readln(which);
   writeln('v1[', which : 1, '] is ', v1[which] : 16)
  end
end.
```

This program initializes the array to ten consecutive square roots and then asks for subscript values to display. Any entered number between 1 and 10 results in a different value displayed, which shows that there really are ten numbers in the array. If you type in a number not in the range of 1 to 10, however, this bug box appears:



When using arrays, it is always a good idea to build safeguards into the program to protect against runtime errors like this. For example, the current version of the array_lab program could be made safer by checking the value of "which" before it is used as a subscript:

```
if (which < 1) or (which > 10) then
 writeln('Sorry -- that number is out of my league.')
else
 writeln('v1[', which : 1, '] is ', v1[which] : 16)
```

Now that we have some practical experience under our belts, let's look at some more general rules for array definition and use. Figure 11-1 shows the form used for defining the simple arrays we'll consider first.

Note the subscript description—the part of the definition that goes inside the square brackets—is, syntactically, any ordinal type description. This subscript description may be—and often is—a subrange of the integer type, but Pascal doesn't restrict you to that form. Nor are there any special rules restricting the base type of the arrays you define. The

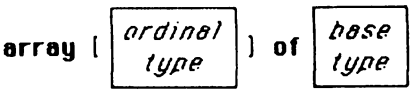


Figure 11-1.

Simple array type definition

following type declaration section contains six legal array definitions:

```
type
 month_type = (JANUARY, FEBRUARY, MARCH, APRIL,
               MAY, JUNE, JULY, AUGUST, SEPTEMBER,
               OCTOBER, NOVEMBER, DECEMBER);
 computer_type = (APPLE, IBM_PC, MACINTOSH, VAX,
                  DEC10, COMMODORE_64);
 chcount_array = array [char] of integer;
 sales_array = array [month_type] of real;
 answer_array = array [Boolean] of string [30];
 tile_array = array [1..7] of 'A'..'Z';
 cost_array = array [computer_type] of computational;
 summer_array = array [JUNE..AUGUST] of longint;
```

Here we have used the enumerated types month_type and computer_type from the previous chapter.

Pascal maintains a general policy of laissez faire toward your type definitions. If you can think up a use for, say, an array [Boolean] of real, Pascal will allow you to define and use it. This flexibility in the design of data structures is one of Pascal's most important features, one you will make use of again and again as you build more sophisticated programs.

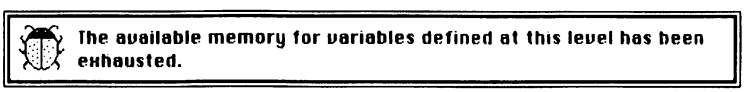
Of course, there are practical limitations on array use. Array variables, like any other variables, consume memory space. Since arrays, by definition, contain an arbitrary number of elements, it is very easy to define large arrays that consume impressive amounts of memory. For example, consider the following program.

```
program array_lab;
{ Experiments on arrays }

const
 ARRAY_SIZE = 10;

type
 array_type = array[1..ARRAY_SIZE] of string;

var
 a : array_type;

begin { array_lab }
 writeln('The array consumes ', sizeof(a) : 1, ' bytes.')
end.
```

Type in and run this program; you should see the output:

```
The array consumes 2560 bytes.
```

This makes sense: the array contains ten strings, and we showed in Chapter 7 that strings require 256 bytes of memory each. To find out how much memory is available for variables in Macintosh Pascal, simply increase the constant definition for ARRAY_SIZE until you get this bug box:



(You may see a slightly different bug box from the one pictured here.)

By experimenting, you can find the largest number for ARRAY_SIZE that will avoid this bug box, which will give you a rough idea of the amount of memory available to Macintosh Pascal programs and variables. (If you have a 512K Macintosh you may notice that sizeof returns a negative "size" for values of ARRAY_SIZE over 127. This is a bug in the Macintosh Pascal version 1.0 that may be fixed in later versions.)

We've just seen that arrays are restricted by the amount of memory available to your program. In some cases, the memory required for array variables may be reduced by

designating the array types to be *packed.* This is done in the type definition simply by preceding the word **array** by the reserved word **packed**, for example:

```
type
 chcount_array = packed array [char] of integer;
 answer_array = packed array [1..30] of Boolean;
 tile_array = packed array [1..7] of 'A'..'Z';
 block = packed array [0..511] of 0..255;
```

You should think of the word **packed** as a request for the array to consume as little space as possible. Whether the array will actually take up less memory as compared to an unpacked array is up to the particular version of Pascal you use. For Macintosh Pascal version 1.0, the packed definition only saves space when the base type of the array is a character or a character subrange. The documentation for Macintosh Pascal claims byte-size integer subranges 0..255 and −128..127 are also packed, but this is not the case in version 1.0. So in the four packed array declarations shown, only the

---

pack(a, i, pa)

arguments:	a - array-type variable i - value of a type compatible with index type of a pa - packed array variable of same base type as a
description:	transfers values from array a starting at a[i] to array pa until pa is filled

unpack(pa, a, i)

arguments:	pa - packed array-type variable i - value of a type compatible with the index type of a
description:	transfers values from array pa to array a starting at a[i] until entire array pa has been copied

Figure 11-2.

Pack and unpack procedure descriptions

third actually results in any saving of memory. (You can verify this yourself by experimenting with the sizeof function.)

Standard Pascal provides two procedures to transfer array element values from packed arrays to unpacked arrays of the same base type, and vice versa; they are called *pack* and *unpack*. Figure 11-2 describes these procedures.

Let's briefly discuss the rules to follow in the use of arrays. In general, unlike the simple types, there is no way to write an array constant, that is, to set all elements of an array to a constant value with a single assignment statement. For example, if v1 is an array of vector_type, as previously defined, the assignment

```
v1 := 0.0
```

is totally illegal; it does not set all elements of v1 to 0. To do that a program must assign each element of the array to 0. One method is to use a **for** loop:

```
for i := 1 to 10 do
 v1[i] := 0.0
```

For similar reasons, it is generally not possible to input or output an entire array using a single read(ln) or write(ln) statement; you must read and write array elements one at a time (assuming, of course, that the array elements are of a type that may be directly read or written).

A program may not compare two entire array variables, even if they are of identical type. Some versions of Pascal allow array comparison but usually only for equality (=) or inequality (<>). For most types of arrays it makes no sense to refer to one array as being "less than" or "greater than" another.

You may pass either entire arrays or individual array elements as arguments to procedures and functions. The normal rules apply concerning agreement of the types of arguments in subprogram calls and the corresponding subprogram definitions. Generally, if you pass a value argument of a certain type, the subprogram should be expecting a value or variable of that same type or a compatible type. If an argument is a variable argument, on the other hand, the types of the expected and actual argument must be identical. You may not pass an element of a packed array as a variable argument.

Often Pascal programmers will choose to pass array arguments as variables rather than values, even when the array variable arguments do not return values from the subprogram (which is the normal reason for specifying variable arguments). The reasoning behind this practice is subtle, but worth knowing. Remember, a value argument is copied into a local variable when the subprogram executes. This copying process consumes extra memory; it also takes a small amount of time to make the copy. While these factors are negligible for small arguments (integers, characters, reals, and so forth), they become more noticeable when large arrays must be copied into temporary local variables every time a subprogram executes. For variable arguments, no such copying occurs. The same region of memory is referenced both in the subprogram and in the calling program. The result is a potentially large saving of memory and a relatively small saving of time.

Finally, there is a rule that Pascal functions may not return array-type values. In general, Pascal functions may only return simple types and (for Macintosh Pascal only) **string** types as results.

Some of Pascal's general rules concerning arrays are relaxed for a special kind of array called a *packed string type.* Packed strings are often used in the same fashion as Macintosh Pascal's **string** type. (Be careful. Packed strings are often simply called strings in texts dealing with Standard Pascal. We will refer to them as packed strings to distinguish them from Macintosh Pascal's nonstandard **string** type. This terminology is also followed in the Macintosh Pascal documentation.) Packed strings are defined in the following way:

```
packed array[1..n] of char
```

Here *n* represents a positive integer constant giving the size of the packed string. Rules for packed strings, as opposed to normal arrays, are as follows:

- You may set an entire packed string variable to a string-constant value. The only restriction is that the string constant must contain exactly as many characters as specified in the packed string's definition.
- An entire packed string variable may be written using a single write or writeln statement.

- Packed string variables or constants may be compared for equality (=) and inequality(<>), and for ordering using the other four relational operators (<, >, <=, >=). Comparison gives the same results as string comparison discussed earlier. The only restriction is that the packed string variables or constants being compared must contain the same number of characters.

Some of these rules are illustrated by the following program. Note that the packed string constants are padded with space characters on the right to expand them to the declared length of the packed string:

```
program paoc_lab;
{ Experiments on packed arrays of characters }

const
 WORDSIZE = 10;

type
 word = packed array[1..WORDSIZE] of char;

var
 w1, w2 : word;

begin { paoc_lab }
 w1 := 'apple     ';
 w2 := 'orange    ';
 if w1 < w2 then
  writeln(w1, ' is less than ', w2)
 else if w1 > w2 then
  writeln(w1, ' is greater than ', w2)
 else
  writeln(w1, ' is equal to ', w2)
end.
```

What we are calling packed strings are the only string-type variables available in Standard Pascal. They are useful primarily when you want to write or run a Pascal program that adheres strictly to Standard Pascal. Otherwise, Macintosh Pascal's built-in string type is considerably more flexible and easier to use.

Packed strings and strings are roughly compatible types. You may assign values of a packed string type to string vari-

ables and vice versa, as long as you don't do something obviously wrong, such as trying to stuff a 20-character string into a 10-character array.

For many purposes, you may treat the Macintosh Pascal **string** type as an array of characters. For example, if s is a **string** variable, then s[3] represents the third character in the string. It is an error, however, to use this technique to try to access a character past the current length of the string.

The base type of an array type can be, generally, any other type including another array type; you may easily define an array of arrays or an array of arrays of arrays, and so on. Such arrays are known as *multidimensional* arrays; each level of array nesting is said to add another dimension to the array type. For example, we could define a new type called matrix_type that would hold ten elements of the vector_type we've already discussed:

```
type
  vector_type = array [1..10] of real;
  matrix_type = array [1..10] of vector_type;
```

Alternatively, we could define the matrix type all at once:

```
type
  matrix_type = array [1..10] of array [1..10] of real;
```

Although this bulky definition is legal, Pascal allows us to specify both subscripts within one set of square brackets:

```
type
  matrix_type = array [1..10, 1..10] of real;
```

All three methods are legal in Pascal and using one over another is usually a matter of personal taste or convenience. Once a multidimensional array type has been defined, you may define variables of that type, as in

```
var
  matrix : matrix_type;
```

Even though matrix is an array of arrays, references to it and its elements follow the same rules already presented. First of all, the single identifier

```
matrix
```

is a reference to the entire two-dimensional array containing 100 (10 times 10) real numbers. Since matrix may be considered an array of vector_type values (by the first definition), the array name and subscript combination

```
matrix[i]
```

refers to the *i*th element of matrix, which happens to be a variable of vector_type. To access one of the elements of the vector_type variable, we play the same game of appending a subscript to the variable name:

```
matrix[i][j]
```

Now our identifier refers to a *real* variable, the *j*th component of the vector_type variable which itself was the *i*th component of the matrix_type variable. Once more, Pascal allows a shortcut in referring to elements of multidimensional arrays so that this reference may be rewritten

```
matrix[i, j]
```

Arrays allow easy manipulation of potentially large amounts of data. Consider the problem of counting the occurrences of all different characters within an input text. Without arrays, we would have to maintain an integer counter variable to keep track of how many A's were seen, another one for B's, and so on. With arrays now at our disposal, we can accomplish this task by declaring an array of nonnegative integers containing one element for each possible character:

```
type
 charcount = array [char] of 0..MAXINT;
...
var
 count : charcount;
```

The subrange 0..MAXINT limits us to using only integers 0 and greater. This technique is often used to emphasize to someone reading the program that the variables involved are counters, and may not be negative. In addition, as de-

scribed in Chapter 10, subranges are promises to Pascal that values outside the subrange will not be assigned to the variables in question. As such, they are a useful check on the internal consistency of your program. In pseudo-code a program to count the number of occurrences of characters would look like this:

```
set all elements in counter array to zero
repeat
    get a line of input
    for each character in line
        increment corresponding array element
until an empty line is entered
report results
```

Note we are following our usual method of accepting input lines until the user enters an empty line by just pressing the RETURN key. This can be translated into a short and sweet program:

```
program charfreq;
{ count frequencies of all characters in input text }

 type
  charcount = array[char] of 0..MAXINT;
 var
  line : string;
  count : charcount;
  ch : char;
  i : integer;

begin { charfreq }
 write('This program counts the frequency of ');
 writeln('occurrence of all characters in an input text.');
 write('Enter text one line at a time; enter an empty ');
 writeln('line when done.');
 for ch := chr(0) to chr(255) do
  count[ch] := 0;
 repeat
  readln(line);
  for i := 1 to length(line) do
   begin
    ch := line[i];
    count[ch] := count[ch] + 1
   end
```

```
  until length(line) <= 0;
  writeln('Results:');
  for ch := chr(0) to chr(255) do
   if count[ch] > 0 then
    begin
     write('', ch, '' occurred ', count[ch] : 1, ' time');
     if count[ch] > 1 then
      writeln('s')
     else
      writeln
    end
end.
```

Type in and run this program, testing it on your own input text. (You will want to expand the Text window to see more of the output.) As an exercise, modify this program so that it counts occurrences of letters only, merging the counts for corresponding upper- and lowercase letters together. (Try using the to_lower function from Chapter 9.) For a new program, you might count occurrences of letter pairs, reporting their frequency at the end of the input text. For this program you might use a two-dimensional array defined as follows:

```
type
  pair_count = array['a'..'z', 'a'..'z'] of integer;
```

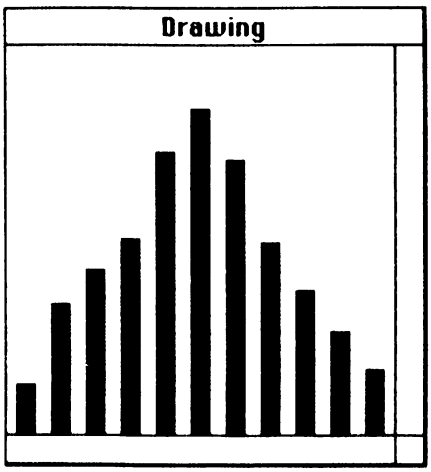
Arrays also make it easier to use more sophisticated graphics effects in your programs. As an example, let's design a program to simulate a large number of rolls of a pair of dice. The program will keep track of how many times each possible value is rolled. Instead of simply printing out a numeric table of the results at the end, however, let's display the results graphically, in histogram (bar graph) format. And better yet, let's display the histogram in real time, as the dice are being rolled, so we can get an idea of how the relative probabilities of each roll change over time.

Possible values resulting from a roll of two dice are the whole numbers from 2 to 12. To keep track of how many times each value occurs, we'll need a counter array type:

```
type
 roll = 2..12;
 roll_array = array[roll] of integer;
```

```
var
 ...
 count : roll_array;
```

In order to display a proper histogram, we'll need to paint 11 different rectangles in the Drawing window. A typical histogram generated during program execution might look like this:



To paint each rectangle bar in the histogram, we will need numbers giving the top, left, bottom, and right sides of the bar. Again, it will be most convenient to keep these numbers in arrays, as follows:

```
var
 top, left, bottom, right : roll_array;
 ...
```

The element

```
right[6]
```

will store the coordinate giving the right edge of the bar representing the number of times the value 6 is thrown. Fortunately, the bottom, left, and right sides of each histogram bar will not change as the program progresses; all we will need to do is to change a bar's top value and repaint the rect-

angle to make it grow upward. As usual, we begin with pseudo-code to set up the overall strategy of the program:

```
initialize bar edges, roll-count array
get total number of throws from user
for each throw
    determine throw value
    increment corresponding count
    update histogram bar length
    if off-scale
        erase and rescale all bars
```

Here we have admitted the possibility that the histogram bars may grow so tall that they will extend outside the Drawing window. We'll handle this problem by maintaining a "scale" variable; the actual pixel height of the bars will be the number of throws divided by the scale variable. Whenever a bar gets too big to be drawn in the window, we'll increase the scale and then erase and redraw the bars using the new scale.

The main program can now be written, assuming we can call on subprograms to do much of the dirty work:

```
program dice_simulation;
{ Simulate a number of throws of two dice }

type
 roll = 2..12;
 roll_array = array[roll] of integer;

var
 top, left, bottom, right : roll_array;
 count : roll_array;
 scale : integer;
 i, numthrows : integer;
 throwval : roll;

{ Insert procedure init_dice_simulation }
{ Insert function randint }
{ Insert procedure update }
{ Insert procedure rescale }

begin { dice_simulation }
 init_dice_simulation;
```

```
 write('Enter number of throws for simulation:');
 readln(numthrows);
 for i := 1 to numthrows do
  begin
   throwval := randint(1, 6) + randint(1, 6);
   count[throwval] := count[throwval] + 1;
   update(throwval, count, top, scale);
   if top[throwval] <= 0 then
    rescale(count, top, scale)
  end
end.
```

We have already seen the randint function in Chapter 9, so we'll only consider the other subprograms here. The first, init_dice_simulation, performs a task common to many Pascal programs: initialization of global variables. With it we can set up the count array to contain zeros and assign the proper values to our histogram bar edges:

```
procedure init_dice_simulation;
{ initialize global variables }

 const
  BAR_OFFSET = 5;
  BAR_WIDTH = 10;
  BAR_SPACING = 18;
  BAR_BOTTOM = 200;

 var
  i : roll;

begin { init_dice_simulation }
 for i := 2 to 12 do
  begin
   count[i] := 0;
   bottom[i] := BAR_BOTTOM;
   left[i] := BAR_OFFSET + (i - 2) * BAR_SPACING;
   right[i] := left[i] + BAR_WIDTH
  end;
 scale := 1
end;
```

The bar positions in the Drawing window are defined using a few constants: BAR_WIDTH is the width of each bar in

pixels, BAR_SPACING is the distance between adjacent bars, BAR_OFFSET is the position of the left edge of the leftmost bar, and BAR_BOTTOM is the coordinate of the bottom edge of the bars. Defining these quantities as constants makes them easy to change and also may aid someone reading the program in understanding how the bars are set up.

The update procedure repaints the proper histogram bar based on the new value thrown. It recalculates a new value for the top of the bar based on the new count for that bar and the current scale, then calls paintrect to draw the new rectangle. The update procedure is written as follows:

```
procedure update (t : roll;
      var count, top : roll_array;
      scale : integer);
{ redraw bar for new throw }

begin { update }
 top[t] := bottom[t] - count[t] div scale;
 paintrect(top[t], left[t], bottom[t], right[t])
end;
```

Finally, the rescale procedure erases all the bars, doubles the scale, and calls update to redraw the bars at the new scale:

```
procedure rescale (var count, top : roll_array;
      var scale : integer);
{ erase, rescale, and redraw histogram bars }

 var
  i : roll;

begin { rescale }
 for i := 2 to 12 do
  eraserect(top[i], left[i], bottom[i], right[i]);
 scale := 2 * scale;
 for i := 2 to 12 do
  update(i, count, top, scale)
end;
```

Try this program using several different values for the total number of throws. If you have some background in

probability you might compare the results from this simulation with the theoretical results. As an exercise, see if you can modify the program to draw horizontal bars from left to right across the Drawing window. As a slightly more difficult problem, modify the program to label each bar with the value of the throw it represents. (Try setting the bottoms of the bars a little higher in the Drawing window.)

RECORDS

Another structured type available in Pascal is the *record*. The rationale behind record types is very similar to that behind array types: the need to collect a number of variables under the same name and handle the variables as a unit. In the previous section, we saw that array types accomplished this function for variables of the same type. We use records for grouping variables of different types.

To define a record type, you must provide Pascal with the following information:

- A name for each variable in the record
- The type of each variable in the record.

That's really all there is to it. A bit of nomenclature is also needed in dealing with records: each individual variable in a record is called a *field* (in contrast to arrays, where each variable was called an element).

For example, suppose you wanted to group all the information pertaining to a checking account transaction in a single place. A sample record definition for such a type might look like this:

```
type
...
 checkbook_rec = record
  check_num : integer;
  trans_month : month_type;
  trans_day : 1..31;
  trans_year : integer;
  payee : string[30];
  cleared : Boolean;
  amount : real
 end;
```

In words, this says variables of type checkbook_rec will contain seven fields:

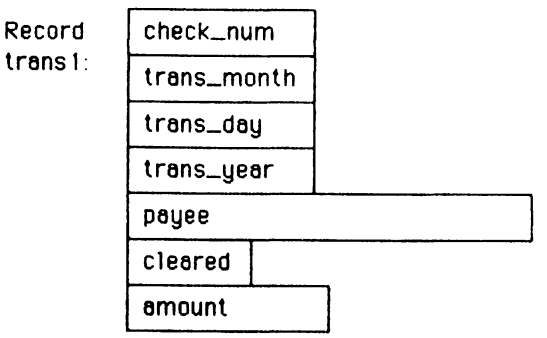
1. Check_num, an integer giving the number of the check used in the transaction.
2. Trans_month, an enumerated type, representing the month of the transaction. (This assumes the definition of the enumerated type month_type precedes the record definition.)
3. Trans_day, an integer subrange 1..31 representing the day of the transaction.
4. Trans_year, an integer giving the year of the transaction.
5. Payee, a string of size 30, representing the person to whom the check is payable.
6. Cleared, a Boolean quantity telling whether the transaction has cleared at the bank or not.
7. Amount, a real number giving the amount of the transaction.

We could probably come up with additional fields, but these will do for our example.

Remember, defining a type does not actually create any variables of that type. Once the type has been defined, however, variable definition works just as before:

```
var
 trans1, trans2 : checkbook_rec;
```

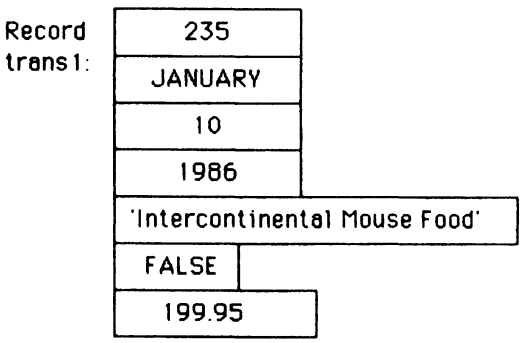
This creates two record variables, trans1 and trans2, of the checkbook_rec type. You might picture record variables as oddly shaped collections of individual variables:



Programs access individual fields of a record variable by specifying the record's name followed by a period and the field's name. The combination of the record name and the field name can be used just as if it were a variable of the type named in the record's type definition. Using the variables just defined, we might record a check written to a Macintosh software supplier as follows:

```
trans1.check_num := 235;
trans1.trans_month := JANUARY;
trans1.trans_day := 10;
trans1.trans_year := 1986;
trans1.payee := 'Intercontinental Mouse Food';
trans1.cleared := FALSE;
trans1.amount := 199.95
```

The net effect of these assignments could be pictured as follows:



Just as we saw with arrays, using the name of the record variable all by itself refers to the entire record. The assignment statement

```
trans2 := trans1
```

copies all the fields from the variable trans1 into the variable trans2. This is a shortcut to assigning each field separately as shown here,

```
trans2.check_num := trans1.check_num;
trans2.trans_month := trans1.trans_month;
trans2.trans_day := trans1.trans_day;
trans2.trans_year := trans1.trans_year;
```

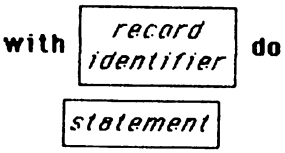


Figure 11-3.

with statement syntax

```
trans2.payee := trans1.payee;
trans2.cleared := trans1.cleared;
trans2.amount := trans1.amount
```

but the net effect is the same in both cases.

You may have noticed that accessing fields in records can be rather long-winded and involves a fair amount of repetitious typing. Pascal provides a statement to help relieve this problem: the **with** statement. The syntax of the **with** statement is shown in Figure 11-3. Informally, the **with** statement is used to specify that all field names in the sub-statement nested within the **with** should be automatically prefixed with the named record identifier.

The previous assignments, for example, could be more succinctly written using a **with**:

```
with trans1 do
begin
 check_num := 235;
 trans_month := JANUARY;
 trans_day := 10;
 trans_year := 1986;
 payee := 'Intercontinental Mouse Food';
 cleared := FALSE;
 amount := 199.95
end
```

Like other statements, **with** statements may be nested to an arbitrary depth. This can be useful when a record contains one or more fields which are themselves records; we will consider an example shortly. To decrease the nesting depth, Pascal allows nested **with** statements in the form

```
with rec1 do
  with rec2 do
    statement
```

to be rewritten in the form

```
with rec1, rec2 do
    statement
```

Let's build some general rules based on the specific example just seen. The syntax sketch for a record type definition is

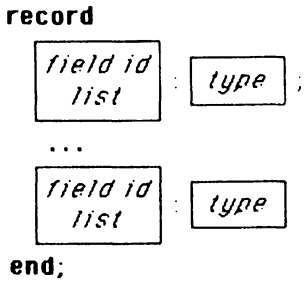


Figure 11-4.

Record type definition syntax

shown in Figure 11-4. Standard Pascal allows the word **packed** to precede the word **record** in a record definition, indicating that variables of the type should consume as little memory as possible. Macintosh Pascal version 1.0 doesn't allow this request, however. (Future versions may.)

Also, there are no particular restrictions on the number or the types of fields you may collect into records. It's a good idea to use record types whenever it makes the data structures used by the program clearer and easier to use. We will discuss a number of examples here, but in general, recognizing appropriate places for using records (or any particular data structure) takes a certain amount of creativity and experience.

Just as we found that arrays of arrays were legal in Pascal, it is also possible to define a record field that is *itself* a record type: records may contain records. For example, in designing a check-transaction record, you might find it cleaner to group the three fields concerning the date into a single record field. This would require, first, defining a record to hold the three values:

```
type
...
 date_rec = record
  month : month_type;
  day : 1..31;
  year : integer
  end;
...
```

Then the simplified check-transaction record could be written as follows:

```
type
...
 checkbook_rec = record
  check_num : integer;
  trans_date : date_rec;
  payee : string[30];
  cleared : Boolean;
  amount : real
 end;
```

Referencing fields of records-within-records is a simple extension of the rules we have already discussed. Assuming that trans1 is a variable of the checkbook_rec type as before, then the reference

```
trans1.trans_date
```

describes a variable of the type date_rec. To access fields of this variable, we simply reapply the "append a period and the field name" rule. For example,

```
trans1.trans_date.month
trans1.trans_date.day
trans1.trans_date.year
```

are the fields containing the month, day, and year of the transaction trans1 refers to. Using these rules together with appropriate **with** statements, the assignments to the fields of the trans1 record could be rewritten as:

```
with trans1 do
 begin
  check_num := 235;
  with trans_date do
   begin
    month := JANUARY;
    day := 10;
    year := 1986
   end;
  payee := 'Intercontinental Mouse Food';
```

```
    cleared := FALSE;
    amount := 199.95
  end;
```

or, equivalently

```
  with trans1, trans_date do
  begin
    check_num := 235;
    month := JANUARY;
    day := 10;
    year := 1986;
    payee := 'Intercontinental Mouse Food';
    cleared := FALSE;
    amount := 199.95
  end
```

Just as there is no special restriction on the types that may become the base type of an array variable, there is no restriction on the types that may be fields in a record. We can even define individual record fields as array types. We may also define an array to be an array of records. Both varieties of data structure are common in Pascal programs.

As an example, let's write a program to simulate shuffling a deck of cards and dealing a bridge hand containing 13 cards from the deck. Each card has two attributes: its rank (Ace, King, Queen, and so on) and its suit. We can group these two attributes into a record definition such as

```
  type
  ...
    card_rec = record
      rank : rank_type;
      suit : suit_type
    end;
```

where the enumerated types suit_type and rank_type have been previously defined as

```
  type
    rank_type = (Two, Three, Four, Five, Six, Seven, Eight,
                 Nine, Ten, Jack, Queen, King, Ace);
    suit_type = (Clubs, Diamonds, Hearts, Spades);
  ...
```

It's natural to consider a deck of cards as an array, since the elements of the deck are all the same type, namely individual cards. The same argument applies to sub-units of the deck, such as bridge hands. In Pascal, we would write:

```
type
...
 deck_array = array [1..DECK_SIZE] of card_rec;
 hand_array = array [1..HAND_SIZE] of card_rec;
```

Here HAND_SIZE and DECK_SIZE are the previously defined constants 13 and 52.

Given this data structure, the program must first initialize the deck, shuffle it, then deal 13 cards from the deck array into the hand array. This is simple enough to write immediately, assuming as always that the details of the process will be hidden away in subprograms:

```
program bridge_deal;
{ Deal a bridge hand from a deck of cards }

 const
  DECK_SIZE = 52;
  HAND_SIZE = 13;

 type
  rank_type = (Two, Three, Four, Five, Six, Seven, Eight,
               Nine, Ten, Jack, Queen, King, Ace);
  suit_type = (Clubs, Diamonds, Hearts, Spades);
  card_rec = record
    rank : rank_type;
    suit : suit_type
  end;
  deck_array = array [1..DECK_SIZE] of card_rec;
  hand_array = array [1..HAND_SIZE] of card_rec;

 var
  deck : deck_array;
  hand : hand_array;

{ Insert procedure init_deck }
{ Insert procedure shuffle }
{ Insert procedure deal }
{ Insert procedure show_hand }
```

```
begin { bridge_deal }
 init_deck(deck);
 shuffle(deck);
 deal(deck, hand);
 show_hand(hand)
end.
```

The first procedure, init_deck, must set up an unshuffled deck. This is done using nested **for** loops and an independent index variable that steps through each card in the deck array:

```
procedure init_deck (var deck : deck_array);
{ initialize deck array }

 var
  i : integer;
  r : rank_type;
  s : suit_type;

 begin { init_deck }
  i := 1;
  for s := Clubs to Spades do
   for r := Two to Ace do
    begin
     deck[i].rank := r;
     deck[i].suit := s;
     i := i + 1
    end
 end;
```

Be assured that no matter how complex and convoluted your data structures, any particular element or field may be accessed by following the consistent and relatively simple rules Pascal provides. Note the method of referring to an individual field of a single record of an array of records. In this program, the variable

```
deck
```

refers to an entire array. To refer to a single element of the array, we must append a subscript within braces:

```
deck[i]
```

This is a reference to a single element of the array deck, which (by the array definition) is a record. To access a field of the record, remember the rule is to append a period and the field name:

```
deck[i].suit
```

Hence the references in the init_deck procedure. (As an exercise, you might try rewriting init_deck using a **with** statement.)

The next routine to consider is shuffle. Shuffling a deck of cards (or any array) is a common problem in game programming; it is also easy to do incorrectly or inefficiently. We need to come up with a random (unpredictable) arrangement or *permutation* of the deck. The algorithm we'll use to generate a random permutation of the deck is simple and fast, although the proof of its correctness is beyond the scope of our discussion. In pseudo-code, the algorithm to shuffle an array containing n elements looks like this:

```
for i := 1 to n - 1:
  set j := random integer between i & n (i ≤ j ≤ n)
  swap element #i and element #j (if i ≠ j)
```

The translation of this into the Pascal procedure shuffle is straightforward:

```
procedure shuffle (var deck : deck_array);
{ shuffle deck of cards }

 var
  i, j : 1..DECK_SIZE;
  c : card_rec;

{ Insert function randint }

begin { shuffle }
 for i := 1 to DECK_SIZE - 1 do
  begin
   j := randint(i, DECK_SIZE);
   if i <> j then
    begin
     c := deck[i];
     deck[i] := deck[j];
```

```
      deck[j] := c
    end
  end
end;
```

To deal the bridge hand from the shuffled cards, we'll just move the first 13 elements of the deck array to the hand array. (If we wanted to be slightly more realistic, we would deal every fourth card from the deck array into the hand array, but since the arrangement of the deck is random, this simpler process shouldn't make any difference.)

```
procedure deal (var deck : deck_array;
      var hand : hand_array);
{ deal cards from deck into hand }

var
 i : 1..HAND_SIZE;

begin { deal }
 for i := 1 to HAND_SIZE do
  hand[i] := deck[i]
end;
```

The last procedure we need is one to display the hand after it has been dealt, show_hand. We'll designate the Drawing window for display:

```
procedure show_hand (var hand : hand_array);
{ display cards in hand }

const
 SPACING = 15;
 OFFSET = 15;

var
 i : 1..HAND_SIZE;

begin { show_hand }
 for i := 1 to HAND_SIZE do
  begin
   moveto(5, (i - 1) * SPACING + OFFSET);
   with hand[i] do
    writedraw(rank, ' of ', suit)
  end
end;
```

The following shows a typical result:

Try out this program. As an exercise, modify it to display all four bridge hands generated from the shuffled deck.

A few miscellaneous but important points about records, similar to the points made in the previous section about general arrays:

- Entire record variables may not be compared with the relational operators, even for equality or inequality. Records must be compared field by field (assuming the fields are comparable).
- Functions may not return record values as results.
- There is no way to write a record constant.
- Records cannot be written or read directly using write(ln) or read(ln) statements.
- Since records often consume relatively large amounts of memory, they are often specified as variable arguments to subprograms even when the subprogram does not affect their values.

VARIANT RECORDS

In the previous section, we saw how records could be used to group together logically related values under a single variable name. While this is a powerful capability, it is sometimes

convenient to be able to define a single record type that may contain different fields. The actual field names and types would vary depending on conditions arising when the program runs. Such records are called *variant records*, and they are special enough to be considered separately.

Let's consider an example first. Suppose you were writing a scholarly paper and you wished to accumulate an impressive list of references. If you wanted to use your computer to help in this task, your first thought might be to design a data structure to hold a reference to a book, as follows:

```
type
...
 book_ref = record
   author : string[40];
   title : string[30];
   edition : integer;
   year_published : integer;
   publisher_name : string[40];
   publisher_city : string[20]
  end;
```

This is suitable for recording book references, but books are only one possible source of information. A reference to an article in a periodical would require some of the same fields, but others (edition, publisher's name and city) would be superfluous, while additional ones would be necessary for recording the name of the periodical, its publication date, and so on. We might write a record for storing periodical references this way:

```
type
...
 periodical_ref = record
   author : string[40];
   title : string[50];
   year_published : integer;
   periodical_name : string[40];
   month_published : month_type;
   first_page, last_page : integer
  end;
```

Keeping two different record types for two different varieties of reference is not an ideal solution, however. One impor-

tant reason is that it would be difficult to write general-purpose, reference-handling subprograms. For example, instead of writing a single procedure to display the information contained in a reference entry, you would need one procedure to display a book-type reference, and another one to display a periodical-type reference. This could easily become intolerable if you needed to keep track of a large number of different reference varieties.

One possible solution is to combine fields for both kinds of reference in a single record definition and to use an enumerated type value to signal which kind of reference is actually contained in the record:

```
type
...
 reference_type = (BOOK, PERIODICAL);

 combined_ref = record
  author : string[40];
  title : string[50];
  reftype : reference_type;
  edition : integer;
  year_published : integer;
  publisher_name : string[40];
  publisher_city : string[20];
  periodical_name : string[40];
  month_published : month_type;
  first_page, last_page : integer
 end;
```

The problem here is, of course, that memory is used inefficiently: records containing book references don't use the fields pertaining to periodicals, and periodical references don't use the book fields. Some of this inefficiency could be alleviated by combining fields together, specifying for example "If this reference is a book, this field is used to hold the publisher's city, but if it is a periodical it holds the name of the periodical." Obviously, this course is fraught with peril and often can't be easily done when the types involved are different.

Pascal's solution is simple, at least when compared to the alternatives. A variant record definition to hold both types of references might appear as follows.

```
reference_rec = record
 author : string[40];
 title : string[50];
 year_published : integer;
 case reftype : reference_type of
  BOOK : (
   edition : integer;
   publisher_name : string[40];
   publisher_city : string[20]
  );
  PERIODICAL : (
   periodical_name : string[40];
   month_published : month_type;
   first_page, last_page : integer
  )
end;
```

The beginning of the record definition follows the same format we discussed previously: a list of field names followed by types. Following this *fixed part* of the record definition comes the *variant part:* the part that differs depending on what we want to store in the record. Note the use of a **case** to distinguish between the two different types of reference. (Don't confuse this use of the word **case** with its use in a **case** statement.) The variants are distinguished by the value of the *tag field* (reftype, in this case).

Informally, this definition says that both types of reference will hold the author's name, a title of the work, and the year of publication. If the reference is a book, the record will also store the publisher's name and city, and the edition of the book. If, on the other hand, the reference is to a periodical, the record will contain the name of the periodical, the month it was published, and the first and last pages of the referenced article. The use of memory will be economized because the same region of memory will be used for storing the fields in the different variants.

Let's assume ref has been declared to be a variable of the reference_rec type. A code segment used to enter information into the variant record might appear as follows:

```
with ref do
 begin
  write('Enter the author(s):');
  readln(author);
```

```
    write('Enter the title:');
    readln(title);
    write('Enter the year of publication:');
    readln(year_published);
    write('Enter the reference type (book or periodical):');
    readln(reftype);
    if reftype = BOOK then
     begin
      write('Enter the edition:');
      readln(edition);
      write('Enter the pubisher''s name:');
      readln(publisher_name);
      write('Enter the publisher''s city:');
      readln(publisher_city)
      end
     else
      begin
      write('Enter the publication month (by name):');
      readln(month_published);
      write('Enter the periodical''s name:');
      readln(periodical_name);
      write('Enter the first page of the article:');
      readln(first_page);
      write('Enter the last page of the article:');
      readln(last_page)
      end
    end
```

Note this program segment only accesses fields applying to periodical references if the ref_kind field has a value other than BOOK, and the book-type fields are only touched if ref_kind is BOOK. It is illegal to refer to a variant field if its variant is not in use.

The syntax previously discussed for record definitions is simply expanded to include variants. Figure 11-5 shows the new syntax with the fixed part of the record the same simple list of names and types we've seen before. The syntax of the variant part is shown in Figure 11-6.

In general, the tag field controlling the variant part may be any ordinal type, and there is no arbitrary limit on the number of different variants you may define in a record. It is an error if the tag field takes on a value not listed as one of the tag constants.

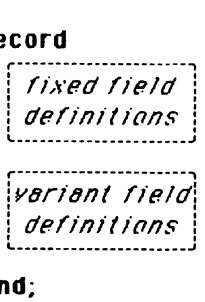


Figure 11-5.

Record type definition syntax (with variants)

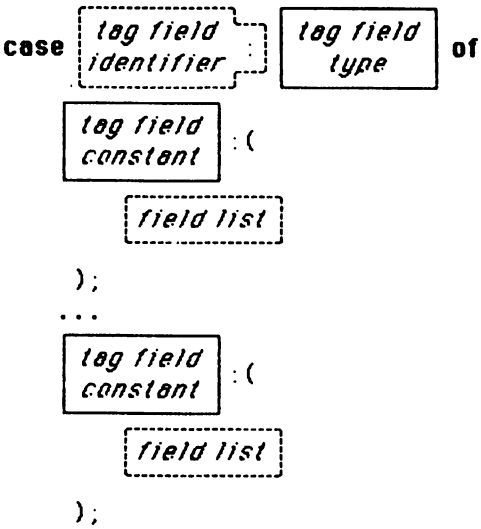


Figure 11-6.

Variant part syntax

Now that we have admitted the possibility of record variants, the following rules should be mentioned:

- Either the fixed part or variant part of a record definition may be absent, but not both.
- If a particular variant has no fields associated with it, an empty pair of parentheses is written after the tag constant.
- Standard Pascal allows variants to be nested. This is not permitted, however, by Macintosh Pascal version 1.0. (Future versions may allow it.)

You'll note in the syntax sketch for the variant part of the record definition that the tag field identifier is optional. When the tag field is omitted, the record has a special name: it is said to be a *free-type union.* Access to any variant in a free-type union can be done without setting a tag field first (since there isn't one). Although such a use is technically illegal, free-type unions can be used as a dodge to subvert Pascal's usually rigorous type checking; the technique is occasionally used by advanced programmers as a last resort

to get around Pascal's rules. As an example, consider the following type definition:

```
type
 fourbytes = record
  case Boolean of
   TRUE : (
    real_val : real
   );
   FALSE : (
    array_val : packed array [1..4] of char
   )
 end;
```

This says variables of the type fourbytes will contain either a single real value or a packed array of four characters. Remember, however, these two different types will occupy exactly the same region of memory since they are both the same size. It is a simple task to write a program that will store a value of one type in the record and retrieve values of the other type:

```
program free_union_lab;
{ an experiment with a free type-union }

type
 fourbytes = record
  case Boolean of
   TRUE : (
    real_val : real
   );
   FALSE : (
    array_val : packed array [1..4] of char
   )
 end;

var
 x : fourbytes;
 i : integer;

begin { free_union_lab }
 while TRUE do
 begin
  write('Enter a real value:');
```

```
    readln(x.real_val);
    write('The equivalent byte array:');
    for i := 1 to 4 do
     write(ord(x.array_val[i]) : 4);
    writeln
   end
end.
```

Try out this program for a number of different real values. (Among other possibilities, you might try consecutive whole-number values or positive and negative powers of 2.) Can you determine what this program is doing? Write a program to perform the inverse operation of accepting four byte values and translating them into the equivalent real numbers.

SETS

Sets are another method of containing different values in a single variable. While arrays contain elements and records contain fields, sets are said to contain *members*. Like arrays, members contained in a set variable must be of the same type (called, once more, the base type of the set). Unlike arrays, however, sets are considered to be unordered with no fixed number of members, although they do have a maximum number of members. The syntax of the set type is shown in Figure 11-7.

The rules for constructing a set type are kept as simple as possible: you need only to specify what values are legal for members of the set. These values must be of an ordinal type, and they must be a contiguous range of values.

Let's consider an example. To declare a set that may contain the positive integer members 1 through 10, one could write the type declaration

```
type
 tenset = set of 1..10;
```

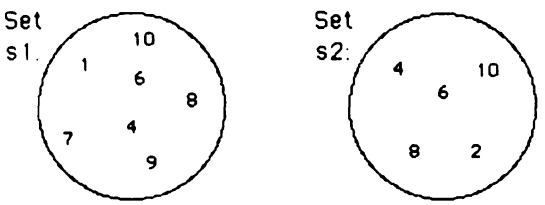
As with any type, you may declare variables of a declared set type:

```
var
 s1, s2 : tenset;
```

Unlike records and arrays, it is possible to write set constants in Pascal. Simply list the elements in the set separated by commas and enclose the list in square brackets. Optionally, the two-period subrange notation (..) may be used as a shorthand to indicate all values from the lower bound of the subrange to the higher bound. For example, the following assignment statements could be used to give values to the declared set variables:

```
s1 := [1, 4, 6, 7..10];
s2 := [2, 4, 6, 8, 10]
```

The effect of these two assignments might be pictured this way:



As already mentioned, the values contained in a set should not be considered to be in any particular order. A set may contain any number of values within the range of its type definition. For example, to place all values between 1 and 10 into the set s1, one would simply write

```
s1 := [1..10];
```

It is also possible for a set to contain no elements at all. This is called, appropriately enough, the *empty set.* To assign s2 the empty set, one would use a pair of empty brackets, as shown:

```
s2 := []
```

set of *base type*

Figure 11-7.

Set type definition syntax

Individual elements in a set constant (including lower and upper bounds on subranges) are, syntactically, expressions evaluated when the program is run. So, for example, this set constant

```
[i, j, i-j..i+j]
```

would be evaluated to contain the values of i, j, and all values between i − j and i + j.

Pascal provides a new relational operator called **in** to test whether a specific value is contained in a set. The expression

```
i in s
```

evaluates to the Boolean value TRUE if the element i is contained in the set s, otherwise it evalutes to FALSE. The **in** operator has the same precedence as the other six relational operators; when evaluated, **in** expects to see a set value on its right and a value compatible with that set's base type on its left.

Some of the operators we've already discussed also apply to set values, although the actual operations implied are different. The operators that may be applied to sets are summarized in Table 11-1. The relative precedence and order of evaluation of these operators remain the same; the set *union* operator + has a lower precedence than the set *intersection* operator * .

Table 11-1.

Set Operators

Operation	Operator	Use	Description
union	+	s1 + s2	result is the set of all elements present in either s1 or s2 (or both)
difference	−	s1 − s2	result is the set of all elements in s1 but not in s2
intersection	*	s1 * s2	result is the set of all elements present in both s1 and s2
subset	<=	s1 <= s2	result is TRUE if all elements in s1 are present in s2, else FALSE
superset	>=	s1 >= s2	result is TRUE if all elements in s2 are present in s1, else FALSE
membership	**in**	e **in** s1	result is TRUE if e is an element of s1, else FALSE

To demonstrate operations on sets, let's write a simple program, set_lab, to initialize two variables of type tenset. It will show you how the various set operations work when applied to the two sets.

```
program set_lab;
{ Experiments on sets }

type
 tenset = set of 1..10;

var
 s1, s2 : tenset;

{ Insert procedure writeset }

begin { set_lab }
 s1 := [1, 4, 6, 7..10];
 s2 := [2, 4, 6, 8, 10];

 writeset(s1);
 write(' + ');
 writeset(s2);
 write(' is ');
 writeset(s1 + s2);
 writeln;

 writeset(s1);
 write(' - ');
 writeset(s2);
 write(' is ');
 writeset(s1 - s2);
 writeln;

 writeset(s1);
 write(' * ');
 writeset(s2);
 write(' is ');
 writeset(s1 * s2);
 writeln;

 writeset(s1);
 write(' = ');
 writeset(s2);
```

```
    write(' is ');
    writeln(s1 = s2);

    writeset(s1);
    write(' <> ');
    writeset(s2);
    write(' is ');
    writeln(s1 <> s2);

    writeset(s1);
    write(' <= ');
    writeset(s2);
    write(' is ');
    writeln(s1 <= s2);

    writeset(s1);
    write(' >= ');
    writeset(s2);
    write(' is ');
    writeln(s1 >= s2)
  end.
```

(Use the Copy and Paste commands to cut down on the repetitive typing involved here.)

The procedure writeset accepts a value (not a variable) of the type tenset and displays it using Pascal's own set notation:

```
procedure writeset (s : tenset);
{ write values in set s }

 var
  i, count : integer;

begin { writeset }
 write('[');
 count := 0;
 for i := 1 to 10 do
  if i in s then
   begin
    if count > 0 then
     write(',');
    write(i : 1);
    count := count + 1
   end;
 write(']')
end;
```

Note that there's no shortcut method for determining what elements are present in a set and which aren't; each possible element must be tested using the **in** operator.

The results you get from running this program should appear as follows (expand the Text window if necessary):

```
[1,4,6,7,8,9,10]  + [2,4,6,8,10] is [1,2,4,6,7,8,9,10]
[1,4,6,7,8,9,10]  - [2,4,6,8,10] is [1,7,9]
[1,4,6,7,8,9,10]  * [2,4,6,8,10] is [4,6,8,10]
[1,4,6,7,8,9,10]  = [2,4,6,8,10] is False
[1,4,6,7,8,9,10] <> [2,4,6,8,10] is True
[1,4,6,7,8,9,10] <= [2,4,6,8,10] is False
[1,4,6,7,8,9,10] >= [2,4,6,8,10] is False
```

Experiment with this program by specifying different elements for s1 and s2 until you get a feeling for how each of the set operations works.

Sets make it possible to solve some problems easily that would be rather difficult without them. For example, given two input strings, consider the problem of determining what characters are present in both strings. Although solvable using other data structures, the solution is easily found with sets. We can define a set type

```
set of char
```

which simply defines a set that can contain all possible character values. We can use two variables of this type to accumulate all distinct characters in the two strings; then the answer to the problem is simply the intersection of the two sets.

Once the data structures have been set up, the program is relatively easy to write:

```
program text_analysis;
{ report characters present in both of two input l

type
 charset = set of char;

var
 cs : array[1..2] of charset;
 line : string;
 ch : char;
 i, j : integer;
```

```
{ Insert procedure write_charset }

begin { text_anaysis }
 for i := 1 to 2 do
  begin
   write('Please enter line #', i : 1, ': ');
   readln(line);
   cs[i] := [];
   for j := 1 to length(line) do
    cs[i] := cs[i] + [line[j]]
  end;
 writeln('Characters in both lines:');
 write_charset(cs[1] * cs[2]);
 writeln;
end.
```

The write_charset procedure is a simple modification of the writeset procedure we saw previously:

```
procedure write_charset (cs : charset);
{ write values in character set cs }

 var
  ch : char;

begin { write_charset }
 for ch := chr(0) to chr(255) do
  if ch in cs then
   write(ch : 2)
end;
```

Try out this program and verify that it works as it should. A simple modification to this program will cause it to print characters present in one string but not in the other, or characters present in both strings. Try writing it and then see if it works.

As a final set example, we'll implement the Sieve of Eratosthenes, a method for generating prime numbers. A prime number is an integer evenly divisible only by itself and 1. The Sieve method begins by considering all numbers between 2 and some maximum value. Since 2 is prime, all multiples of 2 are removed from further consideration (since they are, by definition, divisible by 2). After all multiples of 2 have been removed, the smallest number remaining is 3, which is also

prime. All multiples of 3 are then removed, and then multiples of 5, and so on. At each step after removal of multiples, the smallest number remaining is the next prime.

In psuedo-code, the Sieve algorithm might be written like this:

```
fill sieve with all possible values (2..MAX)
for k := 2 to MAX:
    if k is in the sieve:
        k is prime
        remove all multiples of k from sieve
```

In Pascal, this becomes

```
program Eratosthenes;
{ Implement Sieve of Eratosthenes algorithm }

const
 MAX = 500;

type
 sieve_set = set of 2..MAX;

var
 sieve : sieve_set;
 i, k : integer;

begin { Eratosthenes }
 sieve := [2..MAX];
 for i := 2 to MAX do
  if i in sieve then
   begin
    writeln(i : 1, ' is prime.');
    for k := 1 to MAX div i do
    sieve := sieve - [k * i]
   end
end.
```

Since the value of MAX is 500, this program finds and prints all primes between 2 and 500.

You may use set constants in your programs without declaring set types or variables. One common use of set constants is in compact Boolean expressions. For example, the expression

```
ch in ['a'..'z', 'A'..'Z']
```

gives the value TRUE if the character ch is an uppercase or lowercase letter, otherwise FALSE. This is considerably more concise than writing the equivalent expression

```
(ch >= 'a') and (ch <= 'z') or (ch >= 'A') and (ch <= 'Z')
```

This approach is especially useful when the values to be tested are not contiguous values. For example, the following tests

```
if (i = 3) or (i = 7) or (i = 99) then
...
while (ch = '.') or (ch = ';') or (ch = 'e') or (ch = 'E') do
...
if (month = SEPTEMBER) or (month = APRIL) or
   (month = JUNE) or (month = NOVEMBER) then
...
```

could all be rewritten as set-membership tests:

```
if i in [3, 7, 99] then
...
while ch in ['.', ';', 'e', 'E'] do
...
if month in [SEPTEMBER, APRIL, JUNE, NOVEMBER] then
...
```

The definition of a set type tells Pascal the maximum number of elements the variables of the type can hold. The absolute maximum on the number of items in a set differs between different versions of Pascal. The Macintosh Pascal documentation states that sets whose base type is outside the range −8192 to 8191 "are not supported," which is a common euphemism for saying they probably won't work.

Sets vary in the amount of memory they consume, depending on their definition. Roughly, you can count on each possible set member to consume one bit of memory, whether it is actually present in the set or not. That one bit value is used to indicate the presence of an element or its absence. For example, the tenset type (**set of** 1..10) used earlier in the chapter consumes 10 bits. As an experiment, use the variant record technique described in the previous section to discover the precise method of set representation in Macintosh Pascal.

MACINTOSH PASCAL STRUCTURED TYPES 12

The last thing one discovers in
writing a book is what to put first.

BLAISE PASCAL

Macintosh Pascal provides a number of built-in or predefined structured types that allow your programs to accomplish even more advanced effects than we've seen so far. In this chapter, we'll discuss many of these types and the built-in procedures and functions provided to manipulate them.

POINTS AND RECTANGLES

Points and rectangles were introduced in Chapter 8. Macintosh Pascal provides predefined record definitions that allow direct examination and manipulation of points and rectangles. Both structures are defined as variant records; consider the definition for the type point:

```
vhselect = (V, H);
point = record
 case integer of
  0 : (
   v : integer;
```

```
    h : integer
  );
  1 : (
    vh : array [vhselect] of integer
  )
end;
```

Note that you do not need to put these definitions into your own programs, any more than you need to put in type definitions for real or Boolean; you may consider it to be already typed in for you by Macintosh Pascal. *Vhselect* is an enumerated type (also predefined) that is used as a subscript type in the point definition.

This point type definition really says nothing more than what you already know: to specify a point somewhere on the screen, you must specify two integers as a horizontal coordinate (h) and a vertical coordinate (v). The variant definition simply gives you a choice as to how to refer to these two numbers contained inside the record. Suppose we have defined a variable of type point:

```
var
  ...
  pt: point;
  ...
```

You may refer to the horizontal coordinate of the point pt as either

```
pt.h
```

or

```
pt.vh[H]
```

Similarly, the vertical coordinate of pt can be accessed either by

```
pt.v
```

or

```
pt.vh[V]
```

addpt(pt1, pt2)	
procedure arguments:	pt1—point value pt2—point variable
description:	adds coordinates in pt1 to the corresponding coordinates in pt2; returns result in pt2.
equalpt(pt1, pt2)	
function type:	Boolean
function arguments:	pt1, pt2—point values
description:	TRUE if pt1 and pt2 contain the same coordinates, otherwise FALSE
setpt(pt, h, v)	
procedure arguments:	pt—point variable h, v—integer values
description:	sets horizontal coordinate of pt to h, vertical coordinate to v
subpt(pt1, pt2)	
procedure arguments:	pt1—point value pt2—point variable
description:	subtracts coordinates in pt1 from the corresponding coordinates in pt2; returns result in pt2.

Figure 12-1.

Macintosh Pascal point-manipulation subprograms

Remember the important distinction made in Chapter 8 between points and pixels: points are infinitely small intersections of the grid lines that define QuickDraw's coordinate plane, while pixels lie between pairs of adjacent grid lines. Pixels are all you actually see on the screen. Important built-in procedures in Macintosh Pascal that work with points are shown in Figure 12-1.

A similar variant record scheme is used to define the data type for rectangles called "rect":

```
rect = record
 case integer of
 0 : (
  top : integer;
```

```
    left : integer;
    bottom : integer;
    right : integer
  );
  1 : (
    topleft : point;
    botright : point
  )
end;
```

Like points, the variant record definition for rectangles allows use of two alternate methods of accessing the fields of a rectangle variable. You may either specify the four grid lines that define the top, left, bottom, and right sides of the rectangle, or you may specify the two points that are at the upper left and lower right of the rectangle. So, for example, the following variable references are all integers that refer to the left-hand grid line of the rectangle r:

```
r.left

r.topleft.h

r.topleft[H]
```

All the procedures described in Chapter 8 that accepted rectangle boundaries in the form

```
(top, left, bottom, right ... )
```

may be called with a single rectangle argument to replace the four integer arguments. For example, these four arguments in a call to frameoval,

```
frameoval(top, left, bottom, right)
```

might be replaced by a single rectangle variable:

```
frameoval(r)
```

assuming the variable r contains the desired coordinates of the enclosing rectangle.

Note: this substitution trick of one rectangle variable for four integers is not always permissible. Unless otherwise

noted, when a procedure or function expects a rectangle argument, you must provide it with the rectangle variable's name, not four integers. When you have the choice, however, the method you choose in your own programs should be the one you find more convenient in a given situation.

Additional built-in procedures and functions useful for manipulation of rectangles are shown in Figure 12-2. All the procedures in Figure 12-2 are useful, but you'll find that some are more useful than others. Probably the most useful is setrect. We can set the fields in a rect-type variable directly with four assignment statements,

```
r.left := 33;
r.right := 46;
r.top := 137;
r.bottom := 159
```

but it's considerably more concise to accomplish the same thing with a single call to setrect:

```
setrect(r, 33, 137, 46, 159)
```

In the same way, we can rewrite the chessboard-drawing program we wrote in Chapter 8 by using a rectangle variable, initialized with setrect, and moving one corner of the rectangle with calls to addpt:

```
program chessboard;
{ draw a chessboard }

 var
  r : rect;
  delta : point;
  i : integer;

begin { chessboard }
 setrect(r, 20, 20, 180, 180);
 delta := r.topleft;
 for i := 1 to 8 do
  begin
   invertrect(r);
   addpt(delta, r.topleft)
  end;
 setrect(r, 20, 20, 40, 40);
```

equalrect(r1, r2)

function result:	Boolean
function arguments:	r1, r2—rect values
description:	returns TRUE if r1 and r2 contain the same coordinates, otherwise FALSE.

emptyrect(r)

function result:	Boolean
function argument:	r—rect value
description:	returns TRUE if the rectangle r is empty (encloses no pixels), otherwise FALSE

insetrect(r, dx, dy)

procedure arguments:	r—rect variable dx, dy—integer values
description:	shrinks or expands rectangle r by moving horizontal coordinates inward by dx, vertical coordinates inward by dy. (Dx and dy may be positive or negative)

offsetrect(r, dx, dy)

procedure arguments:	r—rect variable x, y—integer values
description:	adds dx to rectangle's horizontal coordinates, dy to vertical coordinates. (Dx and dy may be positive or negative)

ptinrect(pt, r)

function type:	Boolean
function arguments:	pt—point value r—rect value
description:	returns TRUE if the pixel to the right and below the point pt is inside the rectangle r, otherwise FALSE.

Figure 12-2.

Macintosh Pascal rectangle-manipulation subprograms

pttoangle(r, pt, theta)

procedure arguments:	r—rect value pt—point value theta—integer variable
description:	returns theta as the angle in degrees of a line from the center of rectangle r to point pt, measured clockwise from "straight up."

pt2rect(pt1, pt2, r)

procedure arguments:	pt1, pt2—point values r—rect variable
description:	sets r to the smallest rectangle that has the points pt1 and pt2 at the corners.

sectrect(r1, r2, r)

function type:	Boolean
function arguments:	r1, r2—rect values r—rect variable
description:	sets r to the rectangle enclosing pixels enclosed by both r1 and r2 (if any). Returns TRUE if rect angles intersect, FALSE if they don't.

setrect(r, left, top, right, bottom)

procedure arguments:	r—rect variable left, top, right, bottom—integer values
description:	sets rectangle r to specified boundary coordinates

unionrect(r1, r2, r)

procedure arguments:	r1, r2—rect values r—rect variable
description:	sets rectangle r to the smallest rectangle that encloses all pixels in both r1 and r2.

Figure 12-2.

Macintosh Pascal rectangle-manipulation subprograms (*continued*)

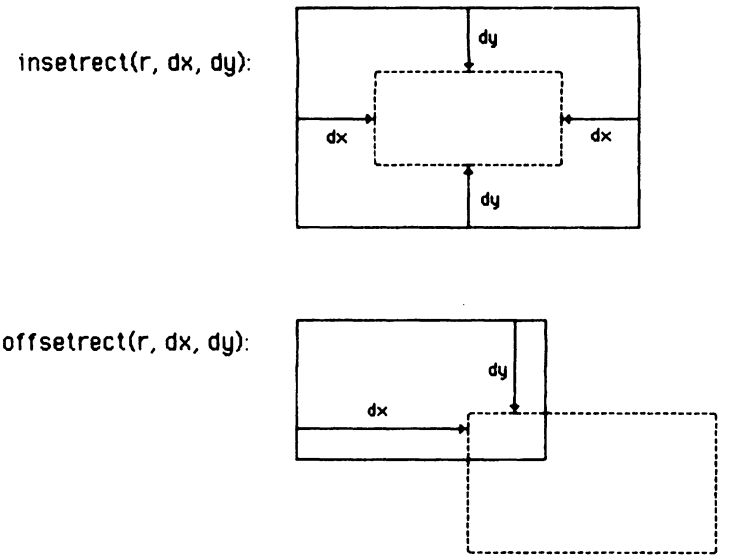


Figure 12-3.

Insetrect and offsetrect

```
for i := 2 to 8 do
 begin
  invertrect(r);
  addpt(delta, r.botright)
 end;
framerect(19, 19, 181, 181)
end.
```

The other routines in Figure 12-2 of above-average usefulness are insetrect and offsetrect. Insetrect is used to shrink or expand rectangles, and offsetrect is used to move them; their actions are diagrammed in Figure 12-3. Note that insetrect and offsetrect don't do any drawing; they only perform mathematical operations on the fields of their rectangle argument. To show their effect, your program must subsequently execute some drawing procedure that uses the altered rectangle. For example, the following flashy program repeatedly inverts a shrinking oval:

```
program flashy_stuff;
{ do some special effects }

 var
  r : rect;

begin { flashy_stuff }
 while TRUE do
  begin
   setrect(r, 0, 0, 200, 200);
   while not emptyrect(r) do
    begin
     invertoval(r);
     insetrect(r, 1, 1)
    end
  end
end.
```

You may have noticed in some of our previous programs that you needed to adjust the output windows in order to see the entire output. It is poor program design to force the user of your programs to initialize window locations and sizes by hand before the program starts. (This is true even when you are the only user of your own programs.) Fortunately, Macintosh Pascal provides simple routines to let your program set up its own window sizes and locations, albeit in a limited way; it allows you to specify whatever arrangement of the Text and Drawing windows you find suitable. Window manipulation procedures available to your program are summarized in Figure 12-4.

The coordinate system used in specifying window rectangles is not the one we've used up until now, which, you'll remember, uses coordinates relative to the upper-left corner of the Drawing window. Instead, the window procedures use the so-called *global* coordinate system which gives positions relative to the upper-left corner of the screen. An easy way to see how this works is with a simple program that hides all windows (removes them from the screen), sets a new location for the Drawing window, then reveals the Drawing window in its new location:

```
program window_lab;
{ experiment with windows }

 var
  r : rect;
```

hideall

procedure arguments:	none
description:	closes (or hides) all windows

getdrawingrect(r)

procedure arguments:	r—rect variable
description:	returns rectangle r giving size and location of the Drawing window

gettextrect(r)

procedure arguments:	r—rect variable
description:	returns rectangle r giving size and location of the Text window

setdrawingrect(r)

procedure arguments:	r—rect value
description:	sets location and size of the Drawing window to the rectangle r

settextrect(r)

procedure arguments:	r—rect value
description:	sets location and size of the Text window to the rectangle r

showtext

procedure arguments:	none
description:	opens (or reveals) the Text window, which becomes the active window.

showdrawing

procedure arguments:	none
description:	opens (or reveals) the Drawing window, which becomes the active window.

Figure 12-4.

Macintosh Pascal window-manipulation procedures

```
begin { window_lab }
 hideall;
 setrect(r, 100, 100, 300, 300);
 setdrawingrect(r);
 showdrawing;
end.
```

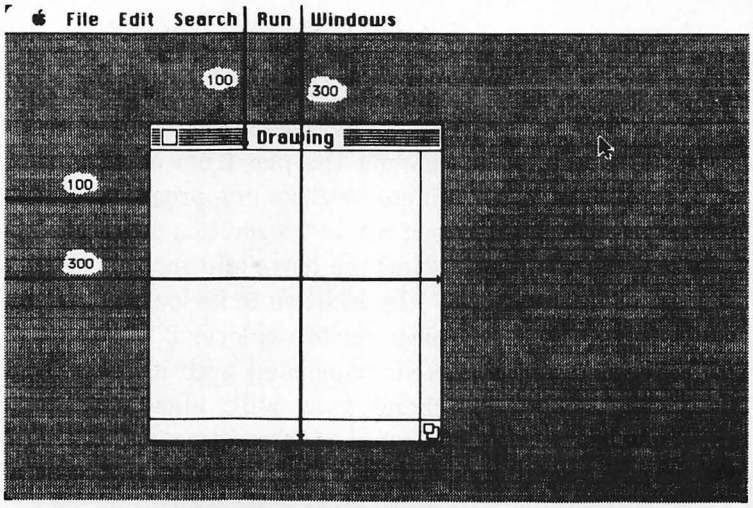


Figure 12-5.

Window placement coordinate system

The result of this program is shown in Figure 12-5; note how the location of the window has been specified in terms of the distance of its edges from the top and left screen edges.

Generally, whenever your program moves or changes the size of the Text or Drawing windows, it should set them back to their previous places afterward. The getdrawingrect and gettextrect procedures allow you to store in rectangle arguments the current locations and sizes of the Drawing and Text windows, respectively. Your program may then move the windows around to whatever positions you find useful; just before the program ends, you then restore the windows to their original position. We've included examples later in the chapter.

THE PEN: PATTERNS, SIZES, AND MODES

Remember from Chapter 8 that a line in QuickDraw is defined to have no thickness. So, like points, QuickDraw lines are invisible and are never actually displayed on the screen.

When QuickDraw draws a line between two points, what is displayed are actually the pixels adjacent to the infinitely thin line, not the line itself.

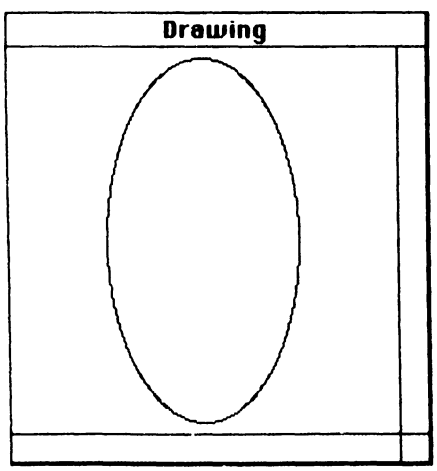
In line drawing, pixels are turned off and on by the QuickDraw pen. We have already seen that since the pen is defined to have a location at all times, to draw a line involves moving the pen from one location to another with the line, lineto, or drawline procedures. You'll also remember that the pen can be moved to a new location without drawing a line by using the move and moveto procedures.

In addition to its location, the pen has other properties we have yet to explore: it has a *size*, a *pattern*, and a *drawing mode* associated with it as well. All these properties may be changed at will, allowing your programs to generate a number of interesting effects.

Before we present the pen-manipulation procedures, let's experiment a bit first. Run the following program:

```
program pen_lab;
{ experiments with the pen }

begin { pen_lab }
 frameoval(5, 50, 195, 150)
end.
```

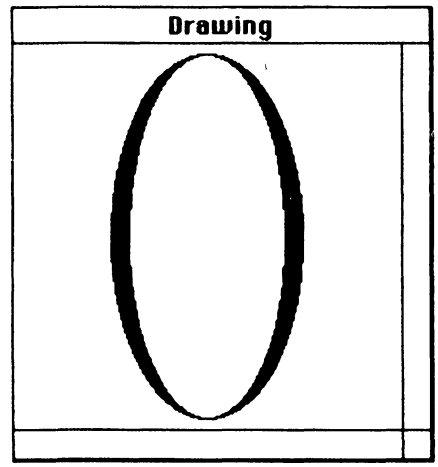
As we've seen before, this simply draws a thin oval in the Drawing window:



To see the effect of changing the pen's size, add the following statement just before the call to frameoval:

```
pensize(10, 1);
```

This time, the oval has a totally different character:



This call to pensize sets the pen to be 10 pixels wide and 1 pixel tall. The result, when framing the oval, is to draw a line 10 pixels wide when the pen moves vertically (along the sides of the oval), a line 1 pixel wide when it moves horizontally (at the oval's top and bottom), and a relatively smooth transition between the two when the pen moves diagonally, similar to a calligraphy pen. The pen's default size—the one that applies if you don't set the size explicitly—is as fine as possible: 1 pixel wide by 1 pixel high.

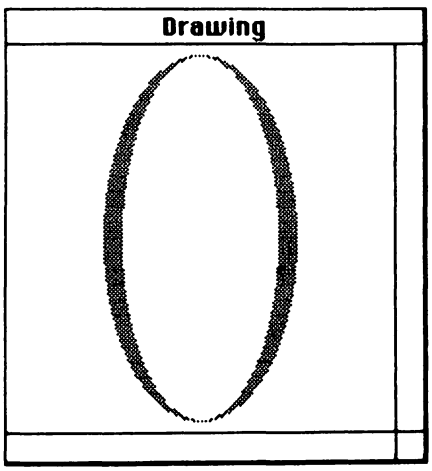
Since the pen's point can get arbitrarily large, it's important to know where it draws compared to the mathematically thin "line" it traverses when moving from one location to another. The rules are simple: the pen's location is defined to be the point at its upper-left corner; the pen hangs below and to the right of this point, affecting only the pixels below and to the right of the mathematical line.

Just as you may change the size and shape of the pen's "point," you may also, figuratively speaking, change the color of the ink it draws with. The ink color is the pen's *pattern.* To change it, one calls the built-in procedure penpat. To see

how it works, add this line to our oval-drawing program just before the call to frameoval:

```
penpat(gray);
```

The result:



The pen now draws in gray. The word "gray" used in the call to penpat is a predefined or built-in variable of the type *pattern.* There are four other predefined patterns available to your programs: ltgray (light gray), dkgrey (dark gray), white, and black. Try substituting these variables in the call to penpat and observe the results.

You are not restricted to using the patterns provided to you by Macintosh Pascal; the pattern type has a perfectly normal (but predefined) type declaration as an array of eight unsigned bytes:

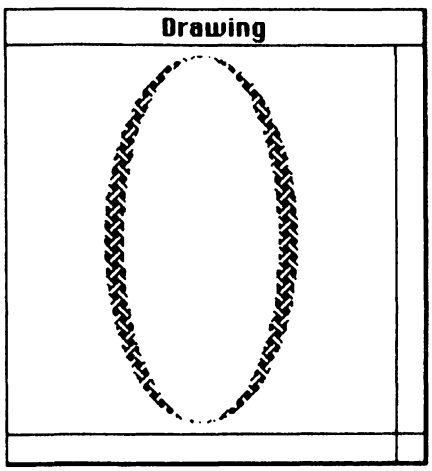
```
pattern = array[0..7] of 0..255;
```

As an example, add a declaration and initializations to pen_lab to use your own pattern variable pat in a line drawing, as follows:

```
program pen_lab;
{ experiments with the pen }

 var
  pat : pattern;
```

```
begin { pen_lab }
 pat[0] := 199;
 pat[1] := 163;
 pat[2] := 17;
 pat[3] := 58;
 pat[4] := 124;
 pat[5] := 184;
 pat[6] := 17;
 pat[7] := 139;
 penpat(pat);
 pensize(10, 1);
 frameoval(5, 50, 195, 150)
end.
```

This results in the following display:



Patterns are based on the underlying binary representation of the eight bytes in the array; each byte represents one row of pixels. Each byte contains eight bits. If a bit's value is 1, the corresponding pixel in the pattern is black; if the bit is 0, the pixel is white. The eight bytes therefore define an eight-by-eight-pixel array. This small pixel group is repeated over and over to generate as large a patterned area as needed. Figure 12-6 shows how the eight bytes are translated into the pixel pattern.

Patterns may also be used independently of the pen. In our previous discussion of QuickDraw, we explored four basic operations: framing, painting, inverting, and erasing. The

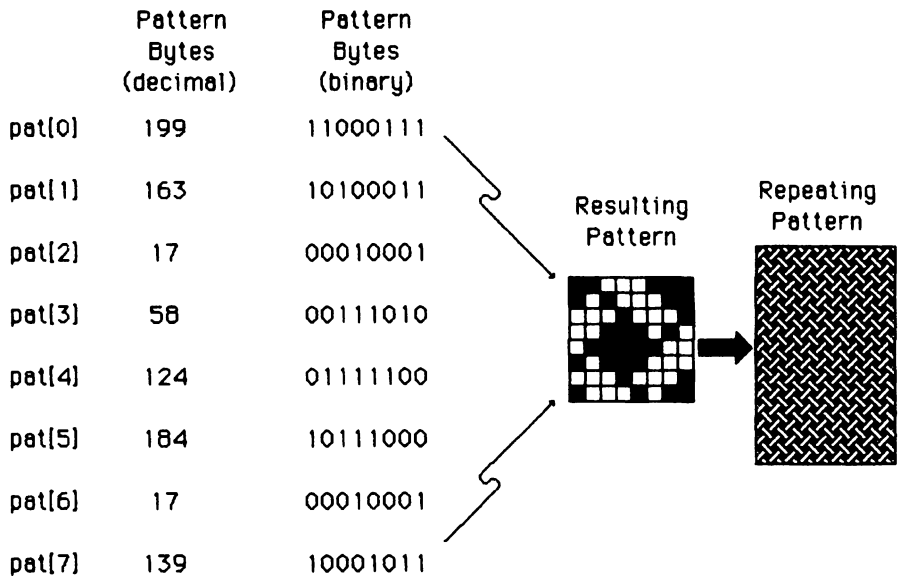


Figure 12-6.

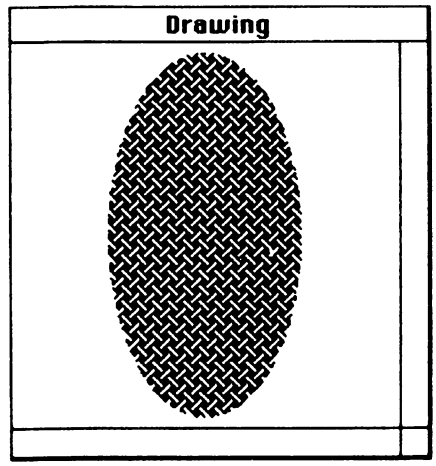
Pattern-to-pixel translation

fifth basic operation QuickDraw performs is *filling:* painting an area with a specified pattern. Calls to filling procedures work similarly to the other shape-drawing procedures; there is simply a trailing pattern argument to the fill procedure. The simple example fills our oval with a pattern:

```
program pattern_lab;
{ experiments with patterns }

var
 pat : pattern;

begin { pattern_lab }
 pat[0] := 199;
 pat[1] := 163;
 pat[2] := 17;
 pat[3] := 58;
 pat[4] := 124;
 pat[5] := 184;
 pat[6] := 17;
```

```
pat[7] := 139;
fillOval(5, 50, 195, 150, pat)
end.
```

The result is the entire oval painted with the basket-weave pattern, which looks like this:



Filling routines for the shapes discussed in Chapter 8 are summarized in Figure 12-7. Note that instead of using the rectangle arguments to these routines, you may replace them with four integer values. You should also recall that the painting operation paints with the current pen pattern and mode, whatever they are; this means you often have a choice between filling and painting whenever you want to display patterned figures. As always, you should make whatever choice is more convenient, understandable, and easy to change.

The final pen attribute we'll consider here in detail is the pen's *mode*. The mode specifies the way the pen turns screen pixels black or white, depending on the pen's pixel pattern and the previous state of the affected screen pixels. There are eight possible modes, numbered 8 through 15. For convenience, Macintosh Pascal (and QuickDraw) defines these numbers as predefined named constants:

```
const
 PATCOPY = 8;
 PATOR = 9;
 PATXOR = 10;
```

fillarc(r, startangle, arcangle, pat)	
procedure arguments:	r—rect value start_angle, arc_angle—integer values pat—pattern value
description:	paints specifed arc with pattern pat
filloval(r, pat)	
procedure arguments:	r—rect value pat—pattern value
description:	paints specified oval with pattern pat
fillrect(r, pat)	
procedure arguments:	r—rect value pat—pattern value
description:	paints rectangle r with pattern pat
fillroundrect(r, oval_wid, oval_ht, pat)	
procedure arguments:	r—rect value oval_wid, oval_ht—integer values pat—pattern value
description:	paints specified rounded rectangle with pattern pat

Figure 12-7.

Macintosh Pascal filling procedures

```
PATBIC = 11;
NOTPATCOPY = 12;
NOTPATOR = 13;
NOTPATXOR = 14;
NOTPATBIC = 15;
```

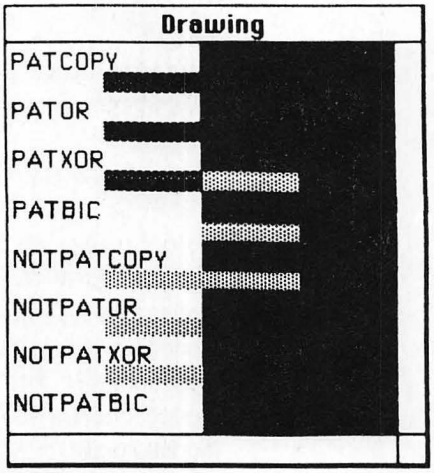
Again, you do not need to make these definitions yourself; they are built in. The pen's drawing mode is changed by calling the procedure penmode and using a single argument to specify the desired mode. The following program shows how modes work with pen-drawing:

```
program mode_lab;
{ experiments with transfer modes }

var
 y, mode : integer;
 modename : array[PATCOPY..NOTPATBIC] of string[10];

{ Insert procedure putstr }

begin { mode_lab }
 modename[PATCOPY] := 'PATCOPY';
 modename[PATOR] := 'PATOR';
 modename[PATXOR] := 'PATXOR';
 modename[PATBIC] := 'PATBIC';
 modename[NOTPATCOPY] := 'NOTPATCOPY';
 modename[NOTPATOR] := 'NOTPATOR';
 modename[NOTPATXOR] := 'NOTPATXOR';
 modename[NOTPATBIC] := 'NOTPATBIC';
 paintrect(0, 100, 200, 200);
 penpat(dkgray);
 pensize(1, 10);
 for mode := PATCOPY to NOTPATBIC do
  begin
   penmode(mode);
   y := 15 + 25 * (mode - PATCOPY);
   putstr(modename[mode], 2, y - 1);
   drawline(50, y, 150, y)
  end
end.
```

This program draws eight dark gray 10-pixel-thick lines; half of each line is drawn against a white background, the other half against a black background. Each line is labeled with the mode in which it was drawn and text output to the Drawing window is accomplished using the putstr routine:

```
procedure putstr (s : string;
      x, y : integer);
{ display string s at (x,y) in Drawing window }

begin { putstr }
 moveto(x, y);
 drawstring(s)
end;
```

The result:



This result gives an intuitive idea of how the various modes work. Of the eight modes, the most important are:

- PATCOPY *paints* the screen with pixels in pattern. The pixels currently on the screen are ignored. The result is simply the black and white pixels in the pattern. (This is the pattern-transfer mode in which Macintosh Pascal programs start.)
- PATOR *overlays* the screen with pixels in a pattern. Black pixels in the pattern cause black pixels to be drawn on the screen, but white pixels in the pattern cause the screen pixels to retain their state.
- PATXOR *inverts* screen pixels corresponding to black pixels in the pattern. White pattern pixels cause no change in screen pixels. The important thing to remember about PATXOR mode is that two successive calls to the same pattern-drawing routine in this mode leaves the screen precisely as before the two calls. This mode is often used to erase a previously drawn pattern on the screen, leaving everything else undisturbed.

The precise effect of all eight modes on different pixel combinations is given in Figure 12-8. (You may want to follow through the tables to see if you can explain the output from the mode_lab program.) Although some of the modes are more useful than others, all have their place in special situations.

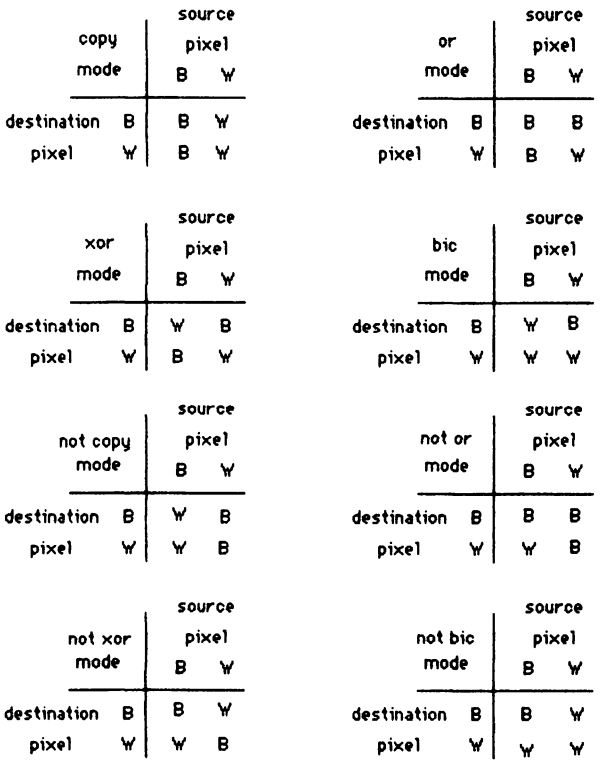


Figure 12-8.

Transfer mode rules

The pen-manipulation routines used by Macintosh Pascal are shown in Figure 12-9. We have already demonstrated how the common operations of changing the pen's mode, size, and pattern are accomplished with calls to penmode, pensize, and penpat, respectively.

Of the remaining routines, perhaps the getpenstate/setpenstate pair deserves some additional attention. These procedures are nearly always called in pairs: first, getpenstate saves the current pen's attributes before the program changes them; the pen's attributes can be changed back to their previous values with a balancing call to setpenstate. The pen's properties are stored in a variable of type penstate, defined this way:

getpen(pt)	
procedure arguments:	pt—point variable
description:	sets pt to current location of pen
getpenstate(state)	
procedure arguments:	state—penstate variable
description:	saves pen's current location, size, mode, and pattern in variable state
penmode(m)	
procedure arguments:	m—integer value
description:	sets pen's drawing mode to m; m should be in the range 8 to 15.
pennormal	
procedure arguments:	none
description:	returns pen to initial state (1 by 1 pixels in size, black pattern, PAT-COPY mode)
penpat(pat)	
procedure arguments:	pat—pattern value
description:	sets pen's drawing pattern to pat
pensize(wid, ht)	
procedure arguments:	wid, ht—integer values
description:	sets pen's size to wid pixels wide and ht pixels high
setpenstate(state)	
procedure arguments:	state—penstate value
description:	restores pen's size, mode, pattern and location to that stored in state

Figure 12-9.

Macintosh Pascal pen-manipulation subprograms

```
penstate = record
   pnloc: point;
   pnsize: point;
   pnmode: integer;
   pnpat: pattern
end;
```

Saving and restoring a pen state would therefore go something like this:

```
var
  ...
  save_state: penstate;


...
getpenstate(save_state);
...
  { calls to penmode, penpat, etc., here }
...
setpenstate(save_state)
...
```

TEXT: STYLES AND MODES

We have already seen in Chapter 8 how to place text in the Drawing window, change to different fonts, and use different point sizes. Macintosh Pascal also allows your programs to display text in different styles: italic, boldface, outlined, and so on. You are probably already familiar with these from MacWrite and MacPaint. This wonderful flexibility in text display is one of the Macintosh's great strengths and fortunately is something which your programs can take advantage of with relative ease.

At this point we need to become slightly more precise in the terminology we've been using and to introduce some new terms. Text drawing starts from the current pen position; the vertical position of the pen establishes a *baseline* for the output characters. Most characters "sit" on the baseline, although characters with descenders (p, g, y, and q, for example) extend below the baseline. Characters may have different widths, depending on the character itself; the letter I is thinner than the letter M, for example. After each character is drawn, the pen moves to the right by the width of the character in preparation for the output of the next character.

The key to displaying different text styles is the definitions of the styleitem and style data types:

```
styleitem = (BOLD, ITALIC, UNDERLINE, OUTLINE,
             SHADOW, CONDENSE, EXTEND);
style = set of styleitem;
```

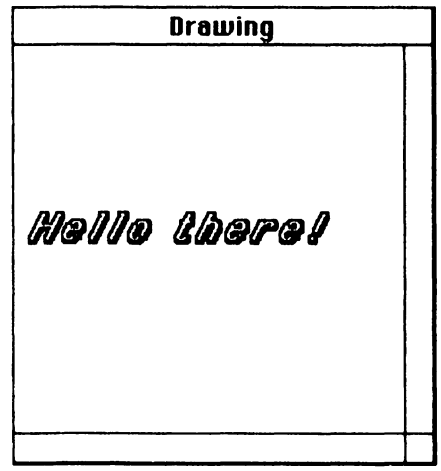
Most of these terms should be familiar to you from MacWrite and MacPaint:

- BOLD—extra pixels are drawn to the right of the character.
- ITALIC—pixels in the character above the baseline are shifted right, pixels below the baseline shifted left.
- UNDERLINE—a line is drawn just underneath the baseline of the character skipping over any descenders.
- OUTLINE—the character is drawn with a hollow interior.
- SHADOW—the character is outlined; the outline is thickened to the right and below to give the appearance of a shadow.
- CONDENSE—horizontal spacing between characters is decreased.
- EXTEND—horizontal spacing between characters is increased.

Any or all of these properties may be specified in a call to the textface routine. Textface accepts a single argument of the style type, and since style is a set of style items, you may simply list the style items you want between square brackets, as a set constant. For example, to print "Hello there!" in shadowed italic bold 18 point Geneva font, one could write

```
program greeting;
{ greet the world }

begin { greeting }
 textsize(18);
 textface([ITALIC, BOLD, SHADOW]);
 moveto(5, 100);
 drawstring('Hello there!')
end.
```

The output from this program is exactly what was requested:



Changing the output style is simply a matter of changing the set of items specified in the call to textface; you may want to experiment with the different possible combinations.

Text output also has its own set of modes. Like pen drawing, mode values are small integers predefined as named constants. Here are their names and values:

```
const
 SRCCOPY = 0;
 SRCOR = 1;
 SRCXOR = 2;
 SRCBIC = 3;
 NOTSRCCOPY = 4;
 NOTSRCOR = 5;
 NOTSRCXOR = 6;
 NOTSRCBIC = 7;
```

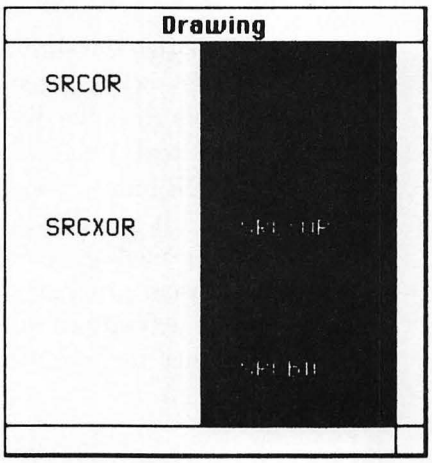
You'll notice the similarity of these names to those of the pen-drawing modes discussed in the previous section. Actually, the same rules we discussed for pen modes apply to similarly named text modes. For example, when drawing text in SRCXOR mode, a black pixel in a character inverts the pixel "under" it on the screen. The most commonly used text-drawing modes are SRCOR, SRCXOR, and SRCBIC. Macintosh Pascal programs start out in SRCOR mode.

The text output mode is set with a call to textmode, which works just like the penmode routine discussed in the previous

sections. The following program is an example of how modes work: the name of each of the three important modes is drawn twice, once against a white background and again against a black background.

```
program mode_lab;
{ experiments with text transfer modes }

 var
  y, mode : integer;
  modename : array[SRCOR..SRCBIC] of string[10];

{ Insert procedure putstr }

begin { mode_lab }
 modename[SRCOR] := 'SRCOR';
 modename[SRCXOR] := 'SRCXOR';
 modename[SRCBIC] := 'SRCBIC';
 paintrect(0, 100, 200, 200);
 for mode := SRCOR to SRCBIC do
  begin
   textmode(mode);
   y := 25 + 75 * (mode - SRCOR);
   putstr(modename[mode], 20, y);
   putstr(modename[mode], 120, y)
  end
end.
```

The output appears as follows:



Once more, try to determine how the transfer mode rules in Figure 12-6 explain the behavior of this program.

Macintosh Pascal also provides routines that allow your program to determine the size of the output text. This is important when a program must place text within certain limits (inside a box, for example). The first of these routines is getfontinfo. A call to getfontinfo has a single argument:

```
getfontinfo(info)
```

Here the variable info must be declared to be of the type fontinfo. The fontinfo type is predefined this way:

```
fontinfo = record
 ascent : integer;
 descent : integer;
 widmax : integer;
 leading : integer
end;
```

The fields have the following meanings:

- ASCENT—the maximum height (in pixels) that a character rises above the baseline.
- DESCENT—the maximum depth (in pixels) that a character descends below the baseline.
- WIDMAX—the width (in pixels) of the widest character in the current font.
- LEADING—the distance (in pixels) between the top of the tallest character and the lowest descender in the line above.

The information retrieved with a call to getfontinfo applies only to the current font, size, and style, of course; when you change any of these, the information you retrieved from getfontinfo becomes obsolete. The information from getfontinfo is useful in obtaining limits on the maximum possible horizontal and vertical extents of a character or a string. It is often useful to find out the actual width in pixels of a character or string. Since most Macintosh fonts contain characters of different widths, this is not as easy as pulling the widmax field of the fontinfo record and multiplying. Macintosh Pascal provides functions that determine the pixel width of a single character (charwidth) and a string of characters (string-

charwidth(ch)

function result:	integer
function arguments:	ch—character value
description:	returns width of ch in pixels (assuming current font, size, and style)

getfontinfo(info)

procedure arguments:	info—fontinfo variable
description:	returns current font ascent, descent, maximum character width, and spacing between adjacent lines

stringwidth(s)

function result:	integer
function arguments:	s—string value
description:	returns width of s in pixels (assuming current font, size, and style)

textface(st)

procedure arguments:	st—style value
description:	sets current style to the attributes found in the set st

textmode(m)

procedure arguments:	m—integer value
description:	sets text transfer mode to m (should be SRCOR, SRCXOR, or SRCBIC)

Figure 12-10.

Macintosh Pascal text-drawing subprograms

width). These routines are shown in Figure 12-10 along with the other text-drawing procedures we've discussed here. We'll see how to use this font information in the next section.

gettime(dt)	
procedure argument:	dt—datetimerec variable
description:	fetches date and time in Macintosh's system clock into dt
settime(dt)	
procedure argument:	dt—datetimerec value
description:	sets Macintosh's system clock to the date and time specified in dt

Figure 12-11.

Macintosh Pascal clock/calendar subprograms

USING MACINTOSH PASCAL STRUCTURED TYPES

In this final section, you'll have a chance to apply the tools discussed so far. Our first program is a digital clock that displays the current date and time continuously.

Pascal provides two built-in routines to access the Macintosh's internal clock, which are described in Figure 12-11. Of these two routines, we'll only be using gettime; while settime might be useful to set the Macintosh's internal date and time from within a program, a more appropriate way is to use the Control Panel or Alarm Clock desk accessories.

Both gettime and settime accept arguments of the type datetimerec. This record type is predefined this way:

```
datetimerec = record
 year, month, day, hour,
 minute, second, dayofweek : integer
 end;
```

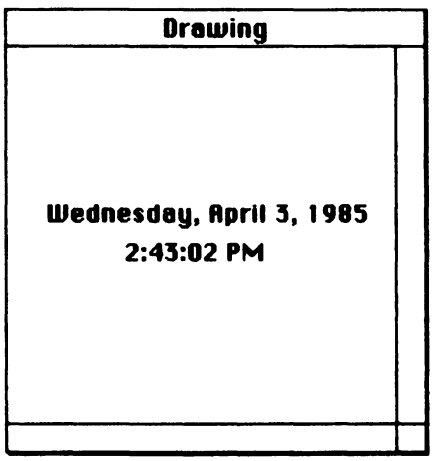
Most fields are self-explanatory. The dayofweek field is an integer signifying what day it is: 1 is Sunday, 2 Monday, and so on. The hour field reports in 24-hour time; possible values are the range of 0 to 23.

A very simple program might call gettime and output the values of the resulting fields:

```
program tell_time;
{ display the current date and time }

var
 dt : datetimerec;

begin { tell_time }
 gettime(dt);
 with dt do
  begin
   writeln('Today"s date: ', month : 1, '/', day : 1, '/', year : 1);
   writeln('The time is : ', hour : 1, ':', minute : 1, ':', second : 1)
  end
end.
```

Type in and run this program to assure yourself that your Macintosh clock is set correctly. (If it isn't, set it using Control Panel or Alarm Clock.)

A slightly more complicated program is one to display the date and time continuously in the Drawing window. What we would like to see is something like this,



where the display changes to reflect the passage of time. A good place to start in our design is in the pseudo-code for the main routine:

```
display current date and time
while mouse button is up
```

```
        erase previous date and time
        display current date and time
    end while
```

The problem with this design is that it results in a program that *continuously* erases and redraws, generating an unpleasant flickering display. (You might want to try writing the program yourself to verify the flickering effect.)

One solution is to update the display only when the time actually moves from one second to another. We will also put the date information on a separate line in the display; this will allow us to change the time display once per second, leaving the date display to be updated only when it actually changes.

A revised pseudo-code might look like this:

```
display current date and time
while mouse button is up do
    get current date and time
    if current time <> last time
        erase old time
        display new time
        if current date <> last date
            erase old date
            display new date
        end if
        last date & time := current date & time
    end if
end while
```

Erasing one string in the window can be accomplished rather easily using the SRCXOR mode discussed in the previous section: writing the same string a second time erases it. We now have enough information to write the main routine:

```
program digital_clock;
{ display date & time continuously }

 const
  DATE_X = 20;
  DATE_Y = 90;
  TIME_X = 60;
  TIME_Y = 110;

 type
  month_name_array = array[1..12] of string[10];
  day_name_array = array[1..7] of string[9];
```

```
var
 dt, old_dt : datetimerec;
 month_name : month_name_array;
 day_name : day_name_array;
 ts, old_ts : string;

{ Insert procedure init_names }
{ Insert function date_string }
{ Insert function time_string }
{ Insert procedure putstr }

begin { digital_clock }
 textmode(patxor);
 textfont(0);
 textsize(12);
 init_names(month_name, day_name);
 gettime(old_dt);
 putstr(date_string(old_dt), DATE_X, DATE_Y);
 old_ts := time_string(old_dt);
 putstr(old_ts, TIME_X, TIME_Y);
 while not button do
  begin
   gettime(dt);
   if dt.second <> old_dt.second then
    begin
     ts := time_string(dt);
     putstr(old_ts, TIME_X, TIME_Y);
     putstr(ts, TIME_X, TIME_Y);
     old_ts := ts;
     if dt.day <> old_dt.day then
      begin
       putstr(date_string(old_dt), DATE_X, DATE_Y);
       putstr(date_string(dt), DATE_X, DATE_Y)
      end;
     old_dt := dt
    end
  end
end.
```

Note the use of the putstr procedure, which was presented earlier in this chapter. Putstr is a general-purpose tool routine that performs the extremely common task of placing a string at a specified point in the Drawing window; we'll be using it in the remainder of our programs.

The arrays month_name and day_name, as you might

expect, are arrays of the names of months and days. The initialization of this array is isolated in a separate procedure:

```
procedure  init_names (var month_name :
      month_name_array;
      var day_name : day_name_array);
{ initialize day and month names }

begin {init_names}
 month_name[1] := 'January';
 month_name[2] := 'February';
 month_name[3] := 'March';
 month_name[4] := 'April';
 month_name[5] := 'May';
 month_name[6] := 'June';
 month_name[7] := 'July';
 month_name[8] := 'August';
 month_name[9] := 'September';
 month_name[10] := 'October';
 month_name[11] := 'November';
 month_name[12] := 'December';
 day_name[1] := 'Sunday';
 day_name[2] := 'Monday';
 day_name[3] := 'Tuesday';
 day_name[4] := 'Wednesday';
 day_name[5] := 'Thursday';
 day_name[6] := 'Friday';
 day_name[7] := 'Saturday'
end;
```

The date_string and time_string functions return the strings corresponding to the date and time represented by their argument. We have isolated these duties into subprograms so things can be easily modified if we want a different formatting for either date or time, or both. Of the two, date_string is the simpler:

```
function date_string (var dt : datetimerec) : string;
{ convert date-time record date info into string }

begin { date_string }
 with dt do
  date_string := stringof(day_name[dayofweek], ', ',
              month_name[month], ' ', day : 1, ', ', year : 1);
end;
```

To translate the numbers contained in the date and time record into a string, date_string uses the stringof function, which was mentioned in Chapter 7. As you may remember, it acts like writeln, except that the formatted output is not displayed but returned as a string.

The time_string function is not that much more difficult; the complications involve displaying either A.M. or P.M. in the string and making sure leading zeros in the minute and second values are displayed. (In other words, displaying "3:04:06" instead of "3: 4: 6".) Here is time_string:

```
function time_string (var dt : datetimerec) : string;
{ convert date and time record into string }

 var
  s : string;
  am_pm_flag : string[2];

begin { time_string }
 with dt do
  begin
   if hour >= 12 then
    begin
     hour := hour - 12;
     am_pm_flag := 'PM'
    end
   else
    am_pm_flag := 'AM';
   s := stringof(hour : 1, ':', minute : 2, ':', second : 2, ' ',
                 am_pm_flag);
   while pos(': ', s) <> 0 do
    s[pos(': ', s) + 1] := '0'
  end;
 time_string := s
end;
```

This completes the digital clock program. Try it out and verify that both date and time are updated correctly.

A natural second choice after programming a digital clock is an analog clock (a clock with hands). Our clock will have three hands: one each for hour, minute, and second. The remainder of the information contained in the date and time record (month, day, and year; day of the week; and whether it is A.M. or P.M.) will be displayed in tiny windows on the clock

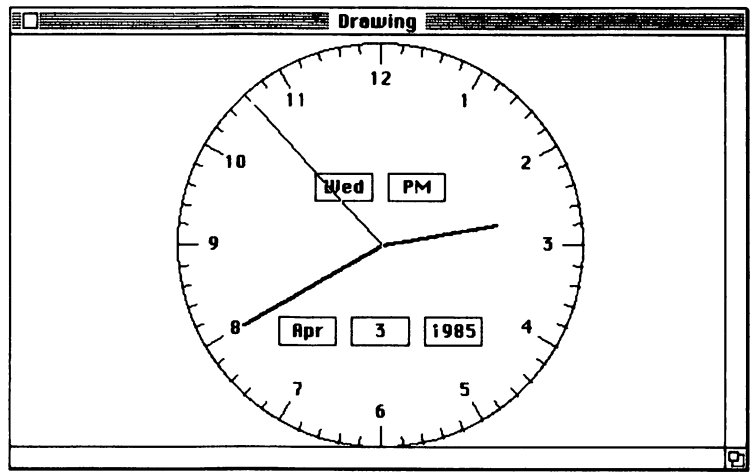


Figure 12-12.

The analog clock

face. When we're done, the result will look something like Figure 12-12. Note this involves initializing the Drawing window to a larger size as discussed in the beginning of this chapter.

The logic involved in keeping the clock up to date is similar to our digital clock program. Every second, the second hand should be erased from its current position and redrawn to point to the next second. When the second hand sweeps over 12, the minute and hour hands are updated to new positions. Finally, twice a day, at noon and midnight, the A.M./P.M. window must be updated; at midnight the date windows must be updated.

The main program, then, looks like this:

```
program analog_clock;
{ display date & time continuously }

const
 PI = 3.14159265;

type
 month_name_array = array[1..12] of string[3];
 day_name_array = array[1..7] of string[3];
```

```
var
 old_dt, dt : datetimerec;
 month_name : month_name_array;
 day_name : day_name_array;
 old_wind, mo_rect, da_rect,
 yr_rect, name_rect, am_pm_rect : rect;
 clock_ctr : point;

{ Insert procedure draw_rad }
{ Insert procedure putstr }
{ Insert procedure center }
{ Insert procedure init_clock }
{ Insert procedure draw_minhr }
{ Insert procedure draw_second }
{ Insert procedure show_day }
{ Insert procedure restore_windows }

begin { analog_clock }
 hideall;
 init_clock(month_name, day_name, old_wind, mo_rect,
da_rect, yr_rect, name_rect, am_pm_rect, clock_ctr);
 gettime(old_dt);
 draw_second(old_dt, clock_ctr);
 draw_minhr(old_dt, clock_ctr);
 show_day(old_dt, mo_rect, da_rect,
          yr_rect, name_rect, am_pm_rect);
 while not button do
  begin
   gettime(dt);
   if dt.second <> old_dt.second then
    begin
     draw_second(old_dt, clock_ctr);
     draw_second(dt, clock_ctr);
     if old_dt.minute <> dt.minute then
      begin
       draw_minhr(old_dt, clock_ctr);
       draw_minhr(dt, clock_ctr);
       if (old_dt.hour <> dt.hour) and
          (dt.hour mod 12 = 0) then
        begin
         show_day(old_dt, mo_rect, da_rect,
                  yr_rect, name_rect, am_pm_rect);
         show_day(dt, mo_rect, da_rect,
                  yr_rect, name_rect, am_pm_rect)
```

```
      end
     end;
    old_dt := dt
   end
  end;
 restore_windows(old_wind)
end.
```

The rectangle variables mo_rect, da_rect, yr_rect, name_rect, and am_pm_rect are simply the locations of the rectangular windows showing the date information.

After hiding all the windows, the analog clock program calls an initializing routine to set up name arrays, initialize the Drawing window, and draw the clock's face. This is a lot to do, and init_clock calls on further nested routines to do each subtask separately:

```
procedure init_clock (
      var month_name : month_name_array;
      var day_name : day_name_array;
      var old_wind, mo_rect, da_rect,
          yr_rect, name_rect, am_pm_rect : rect;
      var clock_ctr : point);
{ initialize global variables }

{ Insert procedure init_names }
{ Insert procedure set_windows }
{ Insert procedure clock_face }
{ Insert procedure set_rects }

begin { init_clock }
 textfont(0);
 textsize(12);
 init_names(month_name, day_name);
 set_windows(old_wind);
 clock_face(clock_ctr);
 set_rects(mo_rect, da_rect, yr_rect,
           name_rect, am_pm_rect);
 textmode(patxor);
 penmode(patxor)
end;
```

Note that both pen drawing and text drawing will be done in XOR mode in this program.

Again, there is an init_names routine to set up the arrays of month and day names:

```
procedure init_names (
      var month_name : month_name_array;
      var day_name : day_name_array);
{ initialize names of months and days }

begin { init_names }
 month_name[1] := 'Jan';
 month_name[2] := 'Feb';
 month_name[3] := 'Mar';
 month_name[4] := 'Apr';
 month_name[5] := 'May';
 month_name[6] := 'Jun';
 month_name[7] := 'Jul';
 month_name[8] := 'Aug';
 month_name[9] := 'Sep';
 month_name[10] := 'Oct';
 month_name[11] := 'Nov';
 month_name[12] := 'Dec';
 day_name[1] := 'Sun';
 day_name[2] := 'Mon';
 day_name[3] := 'Tue';
 day_name[4] := 'Wed';
 day_name[5] := 'Thu';
 day_name[6] := 'Fri';
 day_name[7] := 'Sat'
end;
```

The set_windows procedure expands the Drawing window to nearly the entire screen and hides all others; the old size and location will be saved in the variable old_wind. At the end of the main program, the program calls restore_windows to return the Drawing window to its previous size:

```
procedure set_windows (var old_wind : rect);
{ initialize window display }

 const
  WTOP = 40;
  WBOT = 339;
  WLEFT = 2;
  WRIGHT = 509;
```

```
var
 new_wind : rect;

begin { set_windows }
 getdrawingrect(old_wind);
 setrect(new_wind, WLEFT, WTOP, WRIGHT, WBOT);
 setdrawingrect(new_wind);
 showdrawing
end;


procedure restore_windows (var old_wind : rect);
{ restore original window sizes }

begin { restore_windows }
 hideall;
 setdrawingrect(old_wind);
 showdrawing;
 showtext
end;
```

The set_rects procedure sets up and frames the five small windows in the clock face:

```
procedure set_rects (var mo_rect, da_rect, yr_rect,
                         name_rect, am_pm_rect : rect);
{ initialize display rectangles }

 const
  RECT_WID = 40;
  RECT_HT = 20;
  MO_X = 185;
  MO_Y = 195;
  DA_X = 235;
  DA_Y = 195;
  YR_X = 285;
  YR_Y = 195;
  NAME_X = 210;
  NAME_Y = 95;
  AMPM_X = 260;
  AMPM_Y = 95;

 begin { set_rects }
  setrect(mo_rect, MO_X, MO_Y, MO_X + RECT_WID,
                   MO_Y + RECT_HT);
```

```
  setrect(da_rect, DA_X, DA_Y, DA_X + RECT_WID,
                   DA_Y + RECT_HT);
  setrect(yr_rect, YR_X, YR_Y, YR_X + RECT_WID,
                   YR_Y + RECT_HT);
  setrect(name_rect, NAME_X, NAME_Y,
                     NAME_X + RECT_WID,
                     NAME_Y + RECT_HT);
  setrect(am_pm_rect, AMPM_X, AMPM_Y,
                      AMPM_X + RECT_WID,
                      AMPM_Y + RECT_HT);
  framerect(mo_rect);
  framerect(da_rect);
  framerect(yr_rect);
  framerect(name_rect);
  framerect(am_pm_rect)
end;
```

The clock face is drawn by the clock_face routine. It works from the outside in: first, the large circle is drawn, then 60 small tick marks, followed by the larger 5-minute tick marks, then the numbers 1 to 12:

```
procedure clock_face (var clock_ctr : point);
{ draw clock face }

const
 CLOCK_RADIUS = 140;
 CLOCK_CENTER_X = 255;
 CLOCK_CENTER_Y = 145;
 DIGIT_RADIUS = 115;
 SMALL_TICKS = 7;
 BIG_TICKS = 15;

var
 i, x, y : integer;
 clock_rect, digit_rect : rect;

begin { clock_face }
 setrect(clock_rect, CLOCK_CENTER_X - CLOCK_RADIUS,
                     CLOCK_CENTER_Y - CLOCK_RADIUS,
                     CLOCK_CENTER_X + CLOCK_RADIUS,
                     CLOCK_CENTER_Y + CLOCK_RADIUS);
 setpt(clock_ctr, CLOCK_CENTER_X, CLOCK_CENTER_Y);
 frameoval(clock_rect);
 for i := 1 to 60 do
```

```
 draw_rad(clock_ctr, CLOCK_RADIUS, (i * PI / 30));
 insetrect(clock_rect, SMALL_TICKS, SMALL_TICKS);
 eraseoval(clock_rect);
 insetrect(clock_rect, -SMALL_TICKS, -SMALL_TICKS);
 for i := 1 to 12 do
  draw_rad(clock_ctr, CLOCK_RADIUS, (i * PI / 6));
 insetrect(clock_rect, BIG_TICKS, BIG_TICKS);
 eraseoval(clock_rect);
 insetrect(clock_rect, -BIG_TICKS, -BIG_TICKS);
 for i := 1 to 12 do
  begin
   x := clock_ctr.h
            + round(DIGIT_RADIUS * cos((i - 3) * PI / 6));
   y := clock_ctr.v
           + round(DIGIT_RADIUS * sin((i - 3) * PI / 6));
   setrect(digit_rect, x, y, x, y);
   center(stringof(i : 1), digit_rect)
  end
end;
```

Unfortunately, it takes a little trigonometry to understand everything that's going on here. Adjacent small tick marks are separated by an angle of $360°/60 = 6° = \pi/30$ radians. Similarly, the large tick marks are $360°/12 = 30° = \pi/6$ radians apart. Each set of tick marks is constructed by drawing radial lines from the center of the clock to the edge, then erasing all but the outer parts of the radial lines using insetrect and eraseoval. You may want to set some break points and watch this happen with the debugging aids.

The process of putting numbers on the clock face is also slightly complex. First, the desired location of the number on the clock face is calculated from the distance of the numbers from the center (DIGIT_RADIUS) and the number itself. (Readers with some geometry background may recognize the conversion of polar to rectangular coordinates. In our coordinate system, 0 degrees corresponds to 3 o'clock and angles are measured clockwise.) This calculation gives the desired location of the *center* of the number. The center procedure translates this location into coordinates for the beginning of the number and places the number in the desired spot.

The center procedure actually centers a string within (or, in this case, around) a rectangle. Although the derivation of the correct formula requires a little algebra, the final result is easy to understand.

```
procedure center (s : string;
      var r : rect);
{ center string s in rectangle r }

 var
  info : fontinfo;

begin { center }
 getfontinfo(info);
 with r, info do
  putstr(s, (left + right - stringwidth(s)) div 2,
            (bottom + top + ascent - descent) div 2)
end;
```

Radial lines are drawn with the draw_rad procedure:

```
procedure draw_rad (var center : point;
      r : integer;
      theta : real);
{ draw radial line from center at angle theta, length r }

begin { draw_rad }
 with center do
  drawline(h, v, h + round(r * cos(theta)),
                 v + round(r * sin(theta)))
end;
```

We're done with the initialization routines; the remainder is only a little more work. (If you are typing this in as we go, this is a good time to test what's been done so far. Write stub routines for the other procedures and verify that the clock face gets drawn in a recognizable manner.)

The routine to draw the clock's second hand is a simple call to draw_rad:

```
procedure draw_second (var dt : datetimerec;
      var clock_ctr : point);
{ draw (or erase) second hand }

 const
  SEC_LEN = 130;

 begin { draw_second }
  draw_rad(clock_ctr, SEC_LEN,
                      (dt.second - 15) * PI / 30);
 end;
```

This routine is used to draw a second hand, but since the program has been set to PATXOR mode, redrawing the second hand in the same position will erase the second hand. So this routine does double duty: we are using it for both erasure and drawing. The same comment applies to the draw_minhr routine that draws and erases the minute and hour hands:

```
procedure draw_minhr (var dt : datetimerec;
      var clock_ctr : point);
{ draw (or erase) minute and hour hands }

 const
  MIN_LEN = 110;
  HR_LEN = 80;

begin { draw_minhr }
 pensize(2, 2);
 with dt do
  begin
   draw_rad(clock_ctr, MIN_LEN,
                       (minute - 15) * PI / 30);
   draw_rad(clock_ctr, HR_LEN,
                       (hour - 3 + minute / 60) * PI / 6)
  end;
 pensize(1, 1)
end;
```

Here we've increased the pen's size slightly to emphasize the hour and minute hands, just as on a real clock.

The final routine, show_day, displays the date and A.M./P.M. information in the windows. Again, this routine does double duty: if called a second time with the same parameters, it erases its previous work. (The text transfer mode in effect is SRCXOR.)

```
procedure show_day (var dt : datetimerec;
      mo_rect, da_rect, yr_rect,
      name_rect, am_pm_rect : rect);
{ display (or erase) am/pm and date info }

begin { show_day }
 with dt do
  begin
   if hour >= 12 then
    center('PM', am_pm_rect)
```

```
    else
     center('AM', am_pm_rect);
    center(month_name[month], mo_rect);
    center(stringof(day : 1), da_rect);
    center(stringof(year : 1), yr_rect);
    center(day_name[dayofweek], name_rect);
   end
  end;
```

The calls to the center procedure here are designed to write the indicated strings into the centers of the named rectangles.

Once you get this program working, consider improvements in the appearance of the clock. Could you get it to look more like a real clock? You may want to add sound effects: a tick every second, and perhaps chimes at the hour. How would you make it an alarm clock or a stopwatch?

Our last program may be our most useful. You may have noticed that in our discussion of patterns the numbers corresponding to the basket-weave pattern were, more or less, pulled out of a hat. Changing a pattern, or creating one in the first place, involves a lot of tedious and error-prone binary number manipulations.

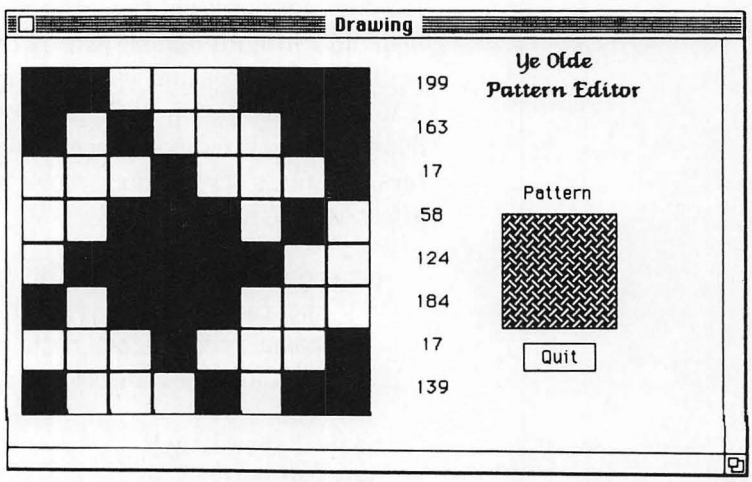


Figure 12-13.

The pattern editor

Fortunately, the Macintosh itself can remove nearly all the drudgery and unreliability involved in creating and modifying patterns. The program we'll develop now is a *pattern editor:* a program that allows you to turn pattern bits on or off and immediately see the effect of the change on the pattern itself. Of course, all binary arithmetic will be done by the Macintosh; we'll also take advantage of the mouse.

Once more, we'll start with a picture of the finished product in operation. Figure 12-13 shows the interaction screen. Here the large 8 by 8 grid on the left is where all editing is done. Each square represents a bit in the pattern, an individual pixel. To change the state of a square from black to white, or vice versa, you simply click in the square with the mouse. Immediately the pattern on the right changes, giving immediate feedback on your change. The numbers to the right of the grid change, too; these are the values of the byte representation of the pattern. The program stops when you click in the Quit rectangle underneath the pattern.

The main routine of the pattern editor appears as follows:

```
program pattern_editor;
{ Allow editing of pattern }

const
 PATROWS = 7;
 PATCOLS = 7;
 GRID_SIZE = 30;
 GRID_X = 10;
 GRID_Y = 20;

type
 row_index = 0..PATROWS;
 col_index = 0..PATCOLS;
 bit_record = record
  r : rect;
  on : Boolean
 end;
 bit_grid_array = array [row_index, col_index] of
                                                bit_record;
var
 grid : bit_grid_array;
 pat_rect, grid_rect, quit_rect : rect;
 old_wind : rect;
 pat : pattern;
```

```
{ Insert procedure putstr }
{ Insert procedure show_grid_bit }
{ Insert procedure show_pat_element }
{ Insert procedure init_pat_edit }
{ Insert function mouse_in }
{ Insert function pressed }
{ Insert procedure check_grid }
{ Insert procedure restore_windows }

begin { pattern_editor }
 hideall;
 init_pat_edit(grid, pat_rect, grid_rect,
               quit_rect, old_wind, pat);
 while not pressed(quit_rect) do
  check_grid(grid, pat_rect, grid_rect, pat);
 restore_windows(old_wind)
end.
```

Note that the procedures putstr and restore_windows have been explained in previous programs; we won't go into them again here. Pay special attention to the definition of the variable grid: it is an 8 by 8 array of records (but the subscripts run from 0 to 7, which you may find slightly confusing). Each element of the grid array contains a rectangle field telling the location of that grid element on the screen. The other field is a Boolean value named "on"; this field tells whether the pixel is black (in which case on is TRUE) or white (on is FALSE).

Once again, much of the work of the program is carried out in the initialization routines. The init_pat_edit procedure mostly calls separate procedures to do the work of setting up windows, titles, the grid, the initial pattern, and the Quit rectangle:

```
procedure init_pat_edit (var grid : bit_grid_array;
     var pat_rect, grid_rect, quit_rect, old_wind : rect;
     var pat : pattern);
{ Initialize global variables, set up initial display }

{ Insert procedure set_windows }
{ Insert procedure draw_titles }
{ Insert procedure init_grid }
{ Insert procedure init_pat }
{ Insert procedure init_quit }
```

```
begin { init_pat_edit }
 textmode(patxor);
 set_windows(old_wind);
 draw_titles;
 init_grid(grid, grid_rect);
 init_pat(pat, pat_rect);
 init_quit(quit_rect)
end;
```

The set_windows routine here is identical to the one used for the analog clock program. Draw_titles simply sets up the labeling of various parts of the screen:

```
procedure draw_titles;
{ Draw titles and labels in window }
 const
  GENEVA = 1;
  VENICE = 5;
  TITLE1_X = 350;
  TITLE1_Y = 20;
  TITLE2_X = 330;
  TITLE2_Y = 40;
  TITLE3_X = 355;
  TITLE3_Y = 110;

begin { draw_titles }
 textfont(VENICE);
 textsize(14);
 putstr('Ye Olde', TITLE1_X, TITLE1_Y);
 putstr('Pattern Editor', TITLE2_X, TITLE2_Y);
 textfont(GENEVA);
 textsize(12);
 putstr('Pattern', TITLE3_X, TITLE3_Y)
end;
```

The init_grid routine initializes the grid (to all white pixels) and draws its representation on the screen. It also initializes the variable grid_rect to the big rectangle enclosing the entire grid:

```
procedure init_grid (var grid : bit_grid_array;
      var grid_rect : rect);
{ initialize and display bit grid }
```

```
var
 row : row_index;
 col : col_index;

begin { init_grid }
 setrect(grid_rect, GRID_X, GRID_Y,
                    GRID_X + (PATCOLS + 1) * GRID_SIZE,
                    GRID_Y + (PATROWS + 1) * GRID_SIZE);
 for row := 0 to PATROWS do
  for col := 0 to PATCOLS do
   with grid[row, col] do
    begin
     setrect(r, col * GRID_SIZE + GRID_X,
                row * GRID_SIZE + GRID_Y,
                (col + 1) * GRID_SIZE + GRID_X,
                (row + 1) * GRID_SIZE + GRID_Y);
     on := FALSE;
     show_grid_bit(grid, row, col)
    end
end;
```

The actual display of a single grid element is handled by the procedure show_grid_bit. Since the grid array contains a field telling the location of the grid element on the screen, this is simply a matter of painting the rectangle black if the corresponding pixel is black, or erasing and framing it if it's white:

```
procedure show_grid_bit (var grid : bit_grid_array;
      row : row_index;
      col : col_index);
{ display single element in bit grid }

begin { show_grid_bit }
 with grid[row, col] do
  if on then
   paintrect(r)
  else
   begin
    eraserect(r);
    framerect(r)
   end
end;
```

The init_pat procedure initializes the pattern we are editing to all white, that is, to all 8 elements having a value of 0; it also sets up the initial display of the actual-size pattern:

```
procedure init_pat (var pat : pattern;
     var pat_rect : rect);
{ initialize and display pattern }

 const
  PAT_X = 340;
  PAT_Y = 120;
  PAT_SQ_SIZ = 80;

 var
  row : row_index;

begin { init_pat }
 for row := 0 to PATROWS do
  begin
   pat[row] := 0;
   show_pat_element(pat, row)
  end;
 setrect(pat_rect, PAT_X, PAT_Y,
                   PAT_X + PAT_SQ_SIZ,
                   PAT_Y + PAT_SQ_SIZ);
 framerect(pat_rect);
 insetrect(pat_rect, 1, 1);
 fillrect(pat_rect, pat)
end;
```

Init_pat calls show_pat_element to display the byte value of each row of the pattern. Since this is a simple numerical output, the procedure is just a single call to putstr:

```
procedure show_pat_element (var pat : pattern;
     row : row_index);
{ Display value of pattern byte }

 const
  BYTE_DISP_X = 280;

begin
 putstr(stringof(pat[row] : 3), BYTE_DISP_X,
          row * GRID_SIZE + GRID_Y + GRID_SIZE div 2)
end;
```

Finally, the init_quit routine sets up and draws the Quit button rectangle and (just so there's no doubt) labels it with the word "Quit." Since we developed the center procedure in the previous program, we'll use it here to center the word inside the rectangle:

```
procedure init_quit (var quit_rect : rect);
{ initialize quit button }

 const
  QUIT_X = 355;
  QUIT_Y = 210;
  QUIT_WID = 50;
  QUIT_HT = 20;

{ Insert procedure center }

 begin { init_quit }
  setrect(quit_rect, QUIT_X, QUIT_Y,
                     QUIT_X + QUIT_WID,
                     QUIT_Y + QUIT_HT);
  framerect(quit_rect);
  center('Quit', quit_rect)
 end;
```

This completes the initialization; as before, this is a good place to get the program running to make sure things behave as they are supposed to.

Returning to the main program, you'll notice that exit from the program is controlled by testing the Boolean function pressed with an argument of quit_rect. In plain language, we are simply trying to say "Keep editing, as long as the Quit rectangle hasn't been clicked." The pressed function is a general-purpose routine, useful in any program where you want to detect pressing.

Although the logic behind the routine is simple, it is also subtle. We consider the Quit rectangle to be pressed when the Mouse button has been both pressed and released while the mouse cursor is inside the button. If the Mouse button is released when the mouse cursor is outside the Quit rectangle, it is not considered to be a press. (However, the mouse can travel outside the Quit button and return any number of times; the only thing that counts is where the cursor is when the Mouse button is released.) This conforms to the Macintosh standard.

Here is the pressed function:

```
function pressed (var r : rect) : Boolean;
{ was button pressed in rectange r ? }

 var
  inside : Boolean;

begin { pressed }
 if not button then
  pressed := FALSE
 else if not mouse_in(r) then
  pressed := FALSE
 else
  begin
   inside := TRUE;
   invertrect(r);
   while button do
    if mouse_in(r) <> inside then
     begin
      invertrect(r);
      inside := not inside
     end;
   if inside then
    invertrect(r);
   pressed := inside
  end
end;
```

Pressed makes use of another Boolean function, mouse_in. A call to mouse_in(r) returns TRUE if the mouse cursor is within the rectangle r, else it returns FALSE:

```
function mouse_in (var r : rect) : Boolean;
{ is mouse inside rectangle r ? }

 var
  mp : point;

begin { mouse_in }
 getmouse(mp.h, mp.v);
 mouse_in := ptinrect(mp, r)
end;
```

If the Quit rectangle is not being pressed, the program

checks for mouse activity in the grid. If the Mouse button is being depressed within the grid, the grid square to which the mouse is pointing is changed to the opposite color (white goes to black, black goes to white). The program then considers that the user is drawing in the grid with that new value. As long as the Mouse button is depressed, the program will set the bits the mouse visits to the same value as the one that was first flipped. (You may recognize this as a rough description of the way Fatbits works in MacPaint.)

The check_grid procedure follows this informal description:

```
procedure check_grid (var grid : bit_grid_array;
      var pat_rect, grid_rect : rect;
      var pat : pattern);
{ check for mouse-editing in bit grid }

 var
  drawval : Boolean;
  row0, row : row_index;
  col0, col : col_index;

{ Insert procedure which_bit }
{ Insert procedure set_bit }

 begin { check_grid }
  if button then
   if mouse_in(grid_rect) then
    begin
     which_bit(row0, col0);
     drawval := not grid[row0, col0].on;
     set_bit(row0, col0, drawval);
     while button do
      if mouse_in(grid_rect) then
       begin
        which_bit(row, col);
        if (row <> row0) or (col <> col0) then
         begin
          row0 := row;
          col0 := col;
          set_bit(row, col, drawval)
         end
       end
    end
 end;
```

Check_grid uses a procedure called which_bit to determine to which grid square the mouse is currently pointing. This is a relatively simple calculation based on the constants that define the grid's left and top edges and the grid-square size:

```
procedure which_bit (var row : row_index;
      var col : col_index);
{ in which grid square is the mouse? }

 var
  x, y : integer;

begin { which_bit }
 getmouse(x, y);
 col := (x - GRID_X) div GRID_SIZE mod (PATCOLS + 1);
 row := (y - GRID_Y) div GRID_SIZE mod (PATROWS + 1)
end;
```

Note that there is a small but real chance that the mouse has scampered outside the grid since the last call to getmouse. We therefore protect the calculation with a mod operation to force the results back into the legal range 0..7.

Finally, a single bit in the pattern is set to its new value by the set_bit procedure. Set_bit sets the grid element to the desired value and displays it by calling show_grid_bit. It then erases the previous byte value for that row by calling show_pat_element.

```
procedure set_bit (row : row_index;
      col : col_index;
      val : Boolean);
{ set bit in pattern to val and display results }

begin { set_bit }
 grid[row, col].on := val;
 show_grid_bit(grid, row, col);
 show_pat_element(pat, row);
 if val then
  pat[row] := bitor(pat[row],
                    bitshift(1, PATCOLS - col))
 else
  pat[row] := bitand(pat[row],
                bitnot(bitshift(1, PATCOLS - col)));
```

```
    show_pat_element(pat, row);
    fillrect(pat_rect, pat)
  end;
```

Single bits in a byte (or, in general, a long integer value) may be set to 0 or 1 without affecting other bits by using the bit functions described in Chapter 7. To set a bit to 1, we can OR the byte with a mask value containing 0 bits everywhere except for a 1-bit at the desired bit position. To set a bit to 0, on the other hand, involves ANDing the byte with a mask containing all 1's and a 0 bit at the desired position. Once the byte has been adjusted, we can display its new value (using show_pat_element again) and display the altered pattern in the pattern box.

That's all we need for the complete pattern editor. After you get it typed in, try to duplicate the basket-weave pattern shown previously. After that, a quick test: what pattern results from the byte sequence 199, 199, 187, 76, 124, 124, 187, 196?

INDEX

E

F

M

N

O

Q

R

S

T

U

V

W

THE FIRST BOOK OF

Macintosh™ Pascal

If you're a beginning Macintosh™ programmer, here is an in-depth introduction to this powerful and versatile language.

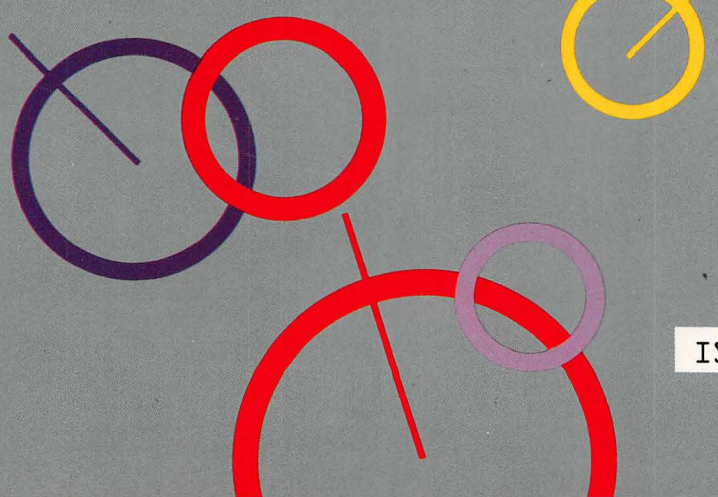
With the lessons contained in this book, you'll learn how to write Pascal programs that utilize the special capabilities of the Macintosh. Through the use of hands-on exercises and numerous examples, you'll be able to write and edit your own useful programs in record time. You'll become an expert at understanding:

- Variables and loops
- Library functions and procedures
- Data types
- Arrays and record sets
- Debugging techniques

Plus more!

For a solid foundation in the essentials of Macintosh Pascal, **The First Book of Macintosh™ Pascal** is the best book you'll ever own.

• Macintosh is a trademark of Apple Computer, Inc.



ISBN 0-07-881165-1