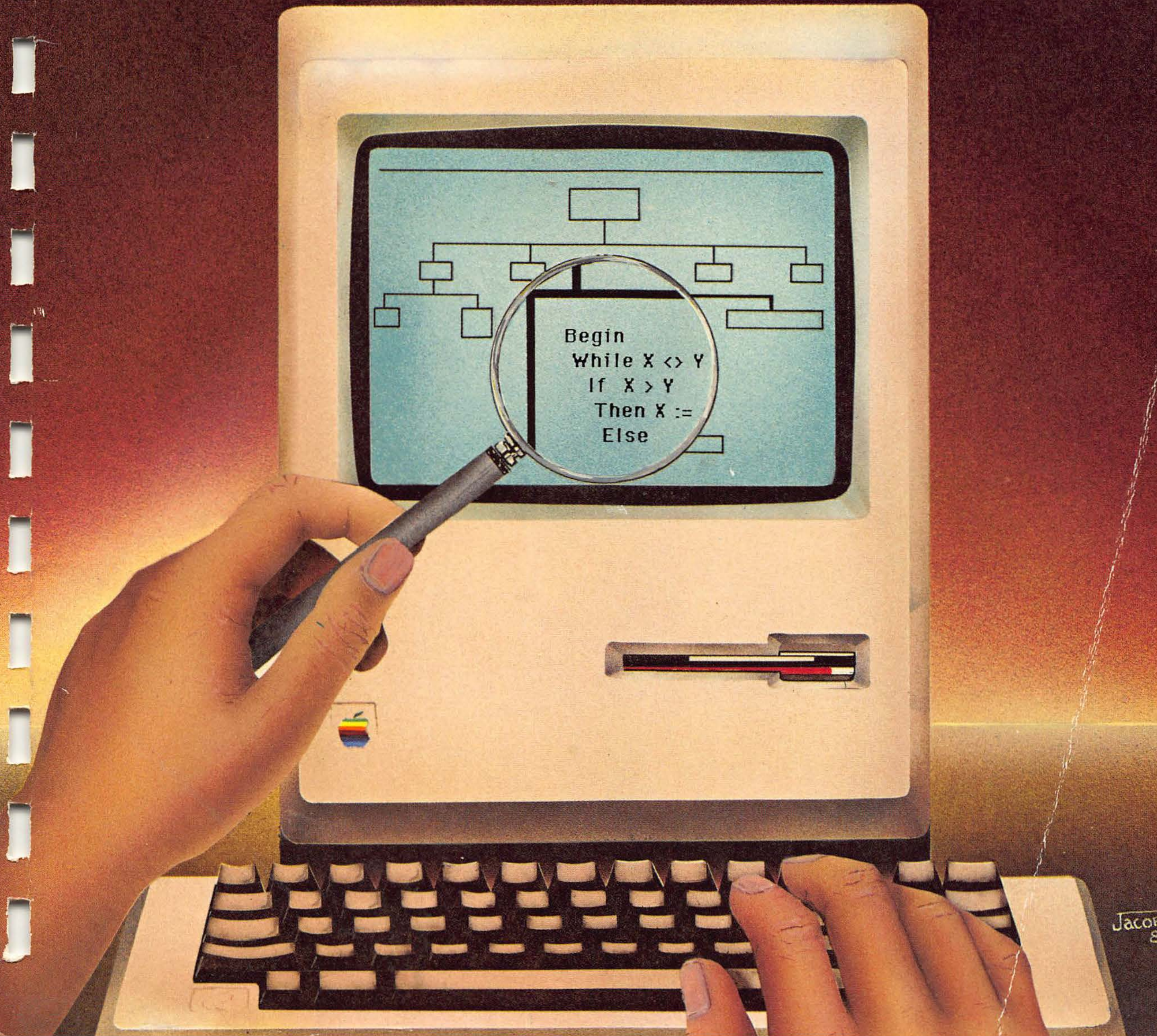
Macintosh™ Hands-On
Pascal

T. G. Lewis
Abbas Birjandi



14–

Don Barns

Macintosh™ Hands-On Pascal

Macintosh™ Hands-On Pascal

Ted G. Lewis
Abbas Birjandi

Wadsworth Publishing Company
Belmont, California
A Division of Wadsworth, Inc.

Production Management: Miller/Scheier Associates
Cover Designer: Detta Penna

Printed in the United States of America

1 2 3 4 5 6 7 8 9 10 90 89 88 87 86

ISBN 0-534-06354-3

Library of Congress Cataloging-in-Publication Data

Lewis, T. G. (Theodore Gyle), 1941-
Macintosh hands-on Pascal.

Includes index.
1. Macintosh (Computer)—Programming. 2. Pascal (Computer program language) I. Birjandi, Abbas. II. Title.
QA76.M3L49 1986 005.265 85-20200
ISBN 0-534-06354-3

Contents

Session 7: Iteration Using While, Repeat, and For 81

Session 8: The Type Statement 91

Session 9: Text Files and Printer Output 99

Preface

This book was inspired by the Macintosh™ and the Macintosh Pascal program, which make learning to program an exciting adventure. There has never been a better hardware and software combination for learning to program in Pascal than the Macintosh Pascal system—including Pascal compilers running on large computers. This powerful combination has advanced the state of the art in both computing and education.

One of the consequences of the use of Macintosh Pascal in education is that it renders textbooks obsolete. In fact, we recommend that you spend very little time reading this or any other book on Pascal. Your time will be better spent trying experiments with Macintosh Pascal, which is why this book has been written as a sequence of hands-on sessions. Each session demonstrates a major concept in programming with Pascal. Furthermore, each session is a slightly more advanced treatment of programming, so you are led into deeper waters by taking small steps.

This book is also organized as a reference to frequently used features. We have included several appendix sections dealing with the syntax and semantics of Pascal, Macintosh, and programming in general. Take a look at these sections before you begin reading to familiarize yourself with these handy references.

We would like to thank Molly Lewis, who edited and typed the rough manuscript. Michele Spear did the reference cards, and several reviewers made many helpful suggestions for improvements.

T.G. Lewis

Abbas Birjandi

Introduction

The Macintosh Personal Computer is the first of a new line of computers to adopt a radical philosophy concerning computers and people. The Macintosh philosophy states that computers should do mundane work and people should do creative thinking. The idea is to make the computer respond to its user according to the user's rules and intuition, rather than bending the human will to conform to computer systems.

You are probably familiar with the visual environment of the Macintosh called the *desk top*. The desk top uses visual cues to help you operate Macintosh with a minimum of jargon and even less knowledge of how computers work. Running a Macintosh is more like driving a bus than being an automobile mechanic.

But now you are ready to learn how to program the Macintosh in Pascal. This is a big commitment because programming is more like being an automobile mechanic than like driving a bus. Macintosh Pascal is easier to learn than Pascal on other machines, but there is no substitute for careful planning, meticulous thought, and downright inventiveness when it comes to programming. The desk-top tools of Macintosh Pascal will help enormously, but you must be prepared to spend many hours disciplining yourself in order to become a successful Macintosh Pascal programmer.

Why Pascal?

Pascal is a high-level programming language for teaching beginners how to program. A *high-level language* is an English-like notation for communicating ideas called *algorithms* to a machine. Pascal is particularly well suited to become your first programming language because of the many ways in which Pascal forces you to discipline your thoughts. This discipline becomes increasingly important as you attack more challenging problems with your Macintosh.

What Is a Programming Language?

Algorithms are very detailed, exactingly precise steps for solving problems. Finding the solution to an equation, keeping a list of telephone numbers, or drawing pictures on the screen all require a long list of detailed and precise steps in order for a machine to do them. Most of what is known about computers has been boiled down to a few thousand algorithms.

An algorithm must be encoded in a form that can be understood by a machine before it becomes a program. (A *program*, of course, is a list of instructions which govern the actions of a machine.) Your Macintosh can understand only one kind of encoding: binary numbers. Binary numbers are strings of ones and zeros, as shown below.

```
10110011
```

This pattern might mean "add" or "copy" or some other machine-level operation. Binary strings are grouped together to form *machine language* programs. Unfortunately, even the most powerful computers cannot "understand" any other form of an algorithm.

Over the past 25 years computer scientists have come to the rescue of programmers who do not wish to learn machine language. These scientists have invented many special-purpose notations called *programming languages* which make programming much easier and thousands of times faster to do.

Suppose you want to tell the Macintosh how to calculate the sales tax on an item selling for AMOUNT dollars.

"Compute the sales tax on AMOUNT dollars at five percent tax rate."

The request can be formalized by using a mathematical notation and being careful how you say what you mean.

"Let Sales_Tax Equal Five Percent of Amount"

Going one step further, the sentence can be succinctly written in a pseudo-English and pseudo-mathematical form by using =, +, –, *, and / for equals, add, subtract, multiply, and divide.

"Sales_Tax = 0.05 * AMOUNT"

This brief example illustrates the essential idea of a high-level programming language like Pascal. Most high-level programming languages are a blend of natural language and mathematics. Natural language is easy for humans to understand, and mathematics is easy for a machine to "understand."

You might wonder why English or some other language is not used exclusively. To show why this is not easy to do, consider the following sentence.

"Payment equals amount owed divided by 10 minus the discount."

This sentence is ambiguous because it could have two interpretations:

1. Payment = owed / 10 - Discount
2. Payment = owed / (10 - Discount)

In other words, the amount owed is either divided by 10, or divided by (10-Discount), depending on your interpretation of the sentence. In a programming language, parentheses are used to remove ambiguity.

Now consider the sentence below.

"Be careful of the corners on that round table."

This sentence is perfectly clear: There is no ambiguity due to missing parentheses or the need to group together words belonging to a certain phrase. Unfortunately, again, the sentence is meaningless because we know from experience that round tables do not have corners. A computer, however, does not have human experience. Instead, algorithms written in a high-level language must be very accurate when they describe the "world" to a machine.

In Pascal the "experience" part of a program must be given through a section of the program that describes objects to the Macintosh. If a "round table" is needed by the program, then it must be described beforehand:

```
Var
    ROUND_TABLE : Real ;
```

This is a Pascal data declaration statement which defines an object called ROUND_TABLE as a *variable* that can assume real number values (real numbers are strings of digits that include a decimal point).

Every object in Pascal has an attribute, such as *Real*, called its type. A *type* is a set of values. ROUND_TABLE can take on real numbers and nothing else; hence its type is Real If ROUND_TABLE had been declared as an *Integer*, it could take on only whole number values without decimal points. Languages like Pascal that *require* you to define the types of all objects used in a program are called *strongly-typed* languages.

English will not do as a programming language because it is ambiguous and it does not give a computer enough information about the types of its objects. High-level languages can be made to resemble English, but this resemblance is deceptive. In actuality, a high-level language is far more formal than any natural language. As a consequence, we must study the rules and regulations of most programming languages in order to write even the most elementary programs.

How Does Macintosh "Understand" Pascal?

If a computer can only understand algorithms encoded in binary, then how does Macintosh "understand" English-like Pascal? In fact, Macintosh does not understand a word of Pascal. Instead, a program called Macintosh Pascal is needed before Pascal programs can be run on the machine. The Macintosh Pascal program is a machine language program which interprets Pascal text much like a human interpreter translates English into French while an English person is communicating with a French person.

In computer terminology, an *interpreter* is any program which carries out the instructions of a high-level program directly upon reading each instruction, one at a time. Most computers do *not* interpret Pascal programs the way Macintosh does. In other systems Pascal is converted first into machine language and then the machine language version of your program is run. In Macintosh Pascal the Macintosh Pascal interpreter never converts Pascal into machine language. Instead, Macintosh Pascal immediately does what each Pascal statement tells it to do.

Interpreters are excellent translators to use when you are first learning to program because they quickly tell you when a mistake is made, and they let you run a partially completed program to see what it does. Interpreters are good for experimenting with a program, and as you will later see, they make programming much easier by giving you features not possible without an interpreter.

How to Use This Book

For a short survey of Pascal read Appendix A: "An Overview of Pascal" at the end of this book. Specific details of each feature of Pascal are given in the sessions, beginning with Session 1.

The best way to learn to program is to do it. This book takes you through a series of increasingly difficult programs by inviting you to actually run programs on the Macintosh.

This approach starts with specific examples of each concept and then moves to the general case. The reader should prepare for a specific-to-general or hands-on approach to learning Pascal. You might consider using a reference guide in addition to this book in order to obtain general information.

The material should be covered from beginning to end. Each session includes several programs to be run on your Macintosh. Be sure you know how to run these programs before moving on to the next session.

A diskette containing all of the programs and another diskette containing some of the MacPaint figures in this book may be purchased from the authors for a nominal fee if you do not want to enter them into your machine.

Session 1:

Getting Started

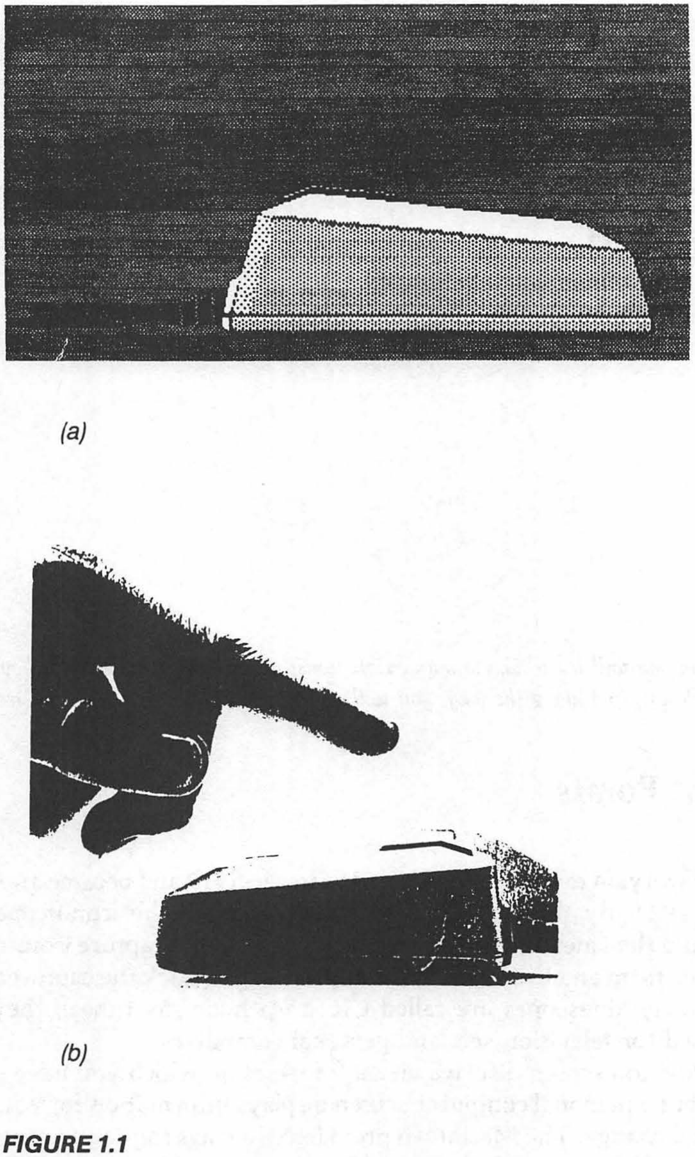
In this session you will learn how to manipulate icons, menus, and windows and how to startup Macintosh Pascal, and along the way, you will become familiar with Macintosh terminology.

The Mouse that Points

Vladimir Zworykin moved to the United States in 1919 and became an American citizen in 1924. By the early 1930s he had invented the iconoscope (image-watcher) and the kinescope. The iconoscope was used to capture icons or images in electronic form and the kinescope was used to play back the captured icons or images. Today kinescopes are called CRTs (cathode ray tubes); they are the screens used for television sets and personal computers.

A television screen displays visual information which you have little control over, but a personal computer screen displays information for you to interact with and change. The Macintosh provides two ways for you to control visual objects on its CRT screen: the keyboard and the mouse. You are probably already familiar with a keyboard, but perhaps this is the first time you have used a mouse. A *mouse* is a hand-controlled pointing and selecting device.

The Macintosh mouse shown in Figure 1.1 consists of a small box with a button on it. The box has a rotating ball beneath it which allows the mouse to be moved around the top of a desk. An electrical wire connects the mouse with the computer so that the computer can sense where the mouse is.



(a)

(b)

FIGURE 1.1
The mouse (a) as Macintosh sees it, and (b) as humans see it.

The signals coming from the mouse are translated by the computer into a screen location. The computer draws a pointer or cursor on its screen corresponding to the location of the mouse on your desk top. The screen pointer moves in concert with the movement of the mouse. Depending on where the mouse is pointing, pressing the mouse button sends a message to the Macintosh which causes some action to be carried out. In Macintosh terminology, pointing with the mouse and pressing its button is called *clicking.*

The button on the mouse may be pressed and released (click), pressed and held down (click-hold), or clicked two times in rapid succession (double-clicking). Furthermore, the mouse may be moved while the button is click-held. This is called *dragging;* it may be used to move a graphical object across the screen or highlight a line of text on the screen.

As you can see, clicking is a simple way to give commands to your computer. The mouse is used just like your finger to point at what you want, but unlike your finger, the mouse can be used to tell the Macintosh what action to perform. Some items can be double-clicked to produce the same result as clicking on two separate items in sequence. This is called a *shortcut* since it is actually just a faster way of doing things. In the following sections we will learn how to give a variety of commands to Macintosh using the mouse.

A Picture Language

One of the major goals of the Macintosh design is to minimize the vocabulary needed to communicate with it by using pictures instead of words to show you what it can do (see Figure 1.2). These graphical cues are called *icons,* or literally, "meaningful symbols." (An icon is defined as "a metaphor or symbol," and in linguistics, it refers to the concept of using pictures rather than letters of the alphabet to communicate ideas.) "Diskette" icons are located in the upper right-hand corner of your screen and naturally look like pictures of little diskettes. The "trash can" icon is a receptacle used for deleted files or programs. These are just two examples of the many and varied icons you will encounter using your Macintosh.

Menus, or, When Is a Computer Like a Restaurant?

When you go to a restaurant, some of the routine actions you perform are to pick up a menu; look at the items under main courses, beverages, and desserts; make your choices; and order them. Macintosh provides you with the same means of communication, except that a Macintosh menu is a list of program options instead of a list of food options like chop suey or french fries.

If you examine a restaurant menu closely, you will notice how foods are grouped together into categories to avoid confusion with foods in other categories. For example, main course items are separated from beverages. Macintosh uses the same idea by displaying the different categories of menu titles at the top of the screen in a region called the *menu bar.*

Unlike a restaurant menu, which displays all of the items in each category of food or drink at once, a Macintosh menu bar conceals the multiple-choice items in each category. The reason for this is not only because the display area of the screen is limited, but, also because displaying all items at once would clutter the screen, making it difficult to find items. Besides, once you've made a

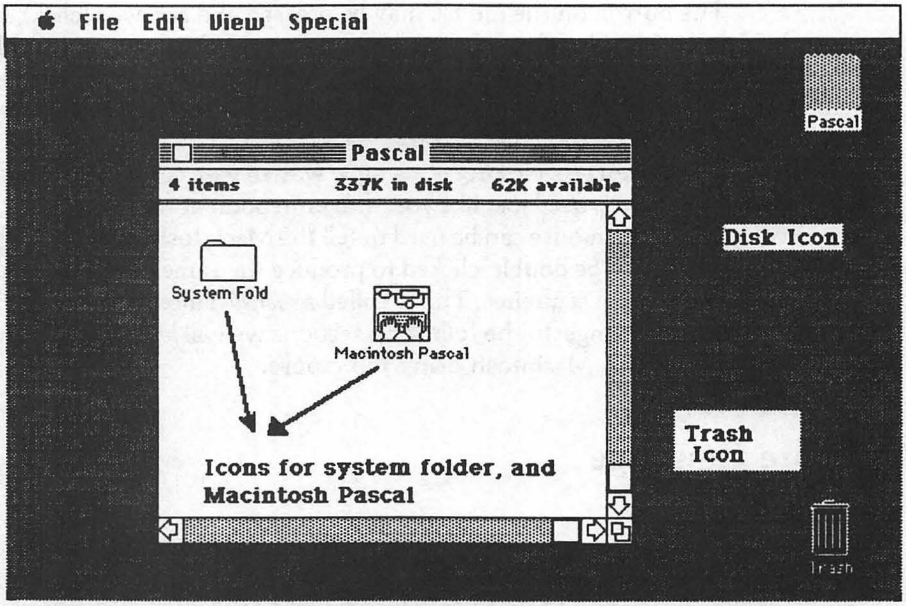


FIGURE 1.2
Some useful icons.

choice in a category you can go on to a new category, ignoring the extraneous detail of previous categories.

Menus are selected by click-holding them; a certain item within a menu is selected by simultaneously pressing the mouse button and moving the mouse until the desired item is chosen. This is called "pulling down" the menu, hence the reason Macintosh menus are called *pull-down menus.* Releasing the button causes the highlighted item to be selected and the menu to disappear.

Throughout this book, we will use the notation "TITLE-ITEM" to mean pulling down a menu TITLE and then selecting an ITEM from it. For example, FILE-OPEN means to pull down menu FILE and then select OPEN.

The pulled-down EDIT menu in Figure 1.3 displays the currently available choices in bold type and the temporarily unavailable (disabled) menu items in dimmed type.

Windows into Macintosh

Suppose you are writing a term paper or a business report. At some time you may want to look at different pages in order to summarize or build other parts of the report. A natural way to do this is to put particular pages next to one

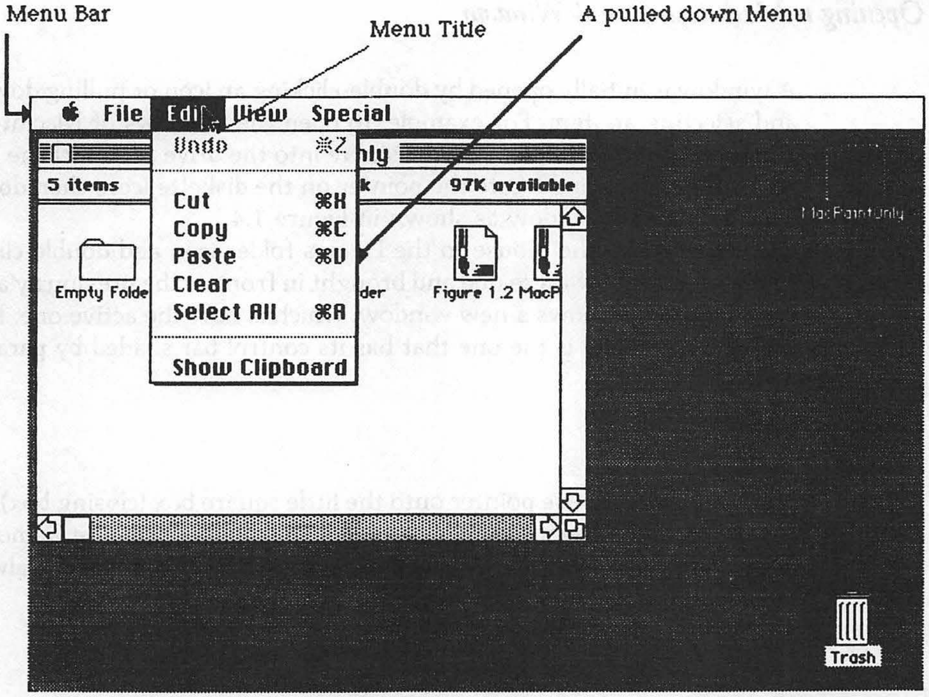


FIGURE 1.3
A pulled-down menu.

another and look at them simultaneously in order to draw information from them for the rest of the report. Each page is like a "window" into the entire report; using multiple pages helps to finish the report in a shorter time.

In a computer system, one or more regions of the screen may be dedicated to different functions. These regions act like pages of reports and can allow you to see portions of several documents at one time. They are called *windows* and are used by the Macintosh to display different pages of information at the same time.

A *window* is any region of the screen which can be moved, re-sized, and scrolled. You can rearrange the position of any window, change its size by making it bigger or smaller, and scroll its contents both horizontally and vertically (see Figure 1.4). A window has: (1) a *size box* to change the window's size, (2) a *scroll bar* for moving through the contents of what is displayed in the window, (3) a *control bar* for moving around the screen, and (4) a *close box* for closing the window.

The *active window* is the one through which you can communicate with the Macintosh at a given instant of time. To make a window active you merely move the mouse pointer to somewhere within its boundaries and click once. Doing so will bring the newly activated window into the foreground and move the previously active one to the background.

Opening a Macintosh Pascal Window

A window is initially opened by double-clicking an icon or pulling down a menu and selecting an item. For example, to open the windows of Macintosh Pascal you insert the Macintosh Pascal diskette into the drive and start the computer. Move the mouse and locate the pointer on the diskette icon, then double-click. This will open a window as shown in Figure 1.4.

Now move the mouse to the Demos folder icon and double-click. Notice that a new window is created and brought in front of the previously active window. Figure 1.5 shows a new window, which is now the active one. In general, the active window is the one that has its control bar shaded by parallel lines.

Closing a Window

Next move the mouse pointer onto the little square box (closing box) in the top left corner of the Demos Window and click the mouse once. The Demos Window disappears. What you did is called *closing* a window. You should always close windows that are no longer needed.

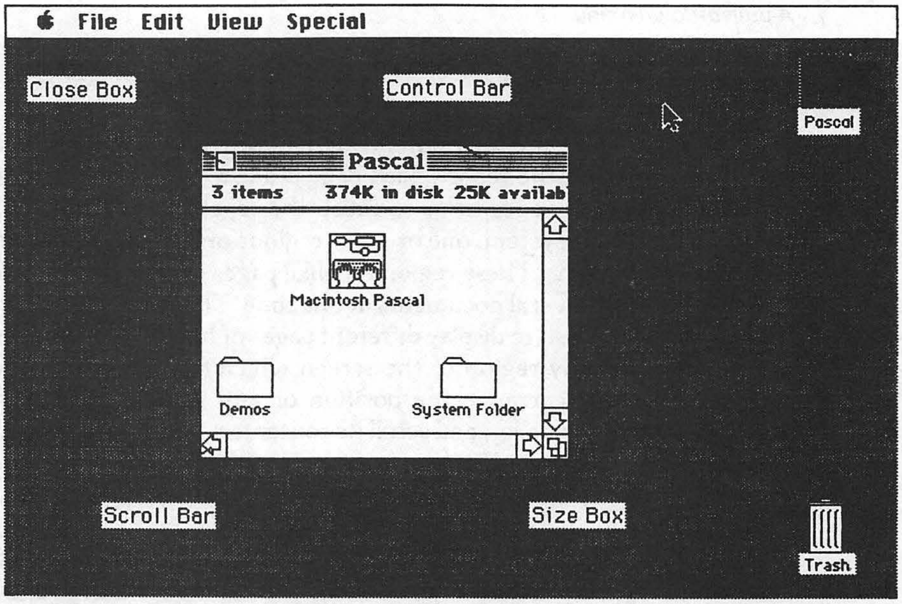


FIGURE 1.4
A window and its parts.

Dragging a Window

The evil forces of chaos may be overcome by occasionally organizing your personal belongings at home or in your desk at the office. Macintosh allows you to clean house or make things more accessible in the same way, by moving objects such as icons and windows about. This capability is called *dragging*. Using the mouse, you can drag an object at will from one location on the screen to another. For example, to move a window point to the control bar at the top of the active window, hold down the mouse button, move (drag) it to another location, and then release the mouse button.

Sizing a Window

The size of a window is changed by "stretching" its lower right corner. Move the mouse to the little square in the lower right corner of any window and drag the corner. Notice how the corner follows the mouse. When the window has been stretched to the desired size, release the mouse button. This is the new size of the active window.

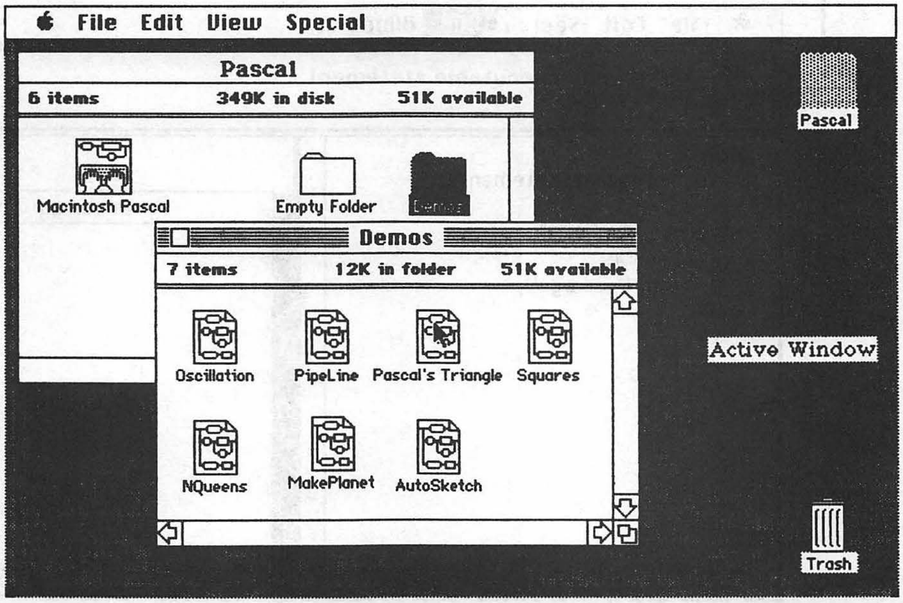


FIGURE 1.5
Pascal and Demos Folder Windows.

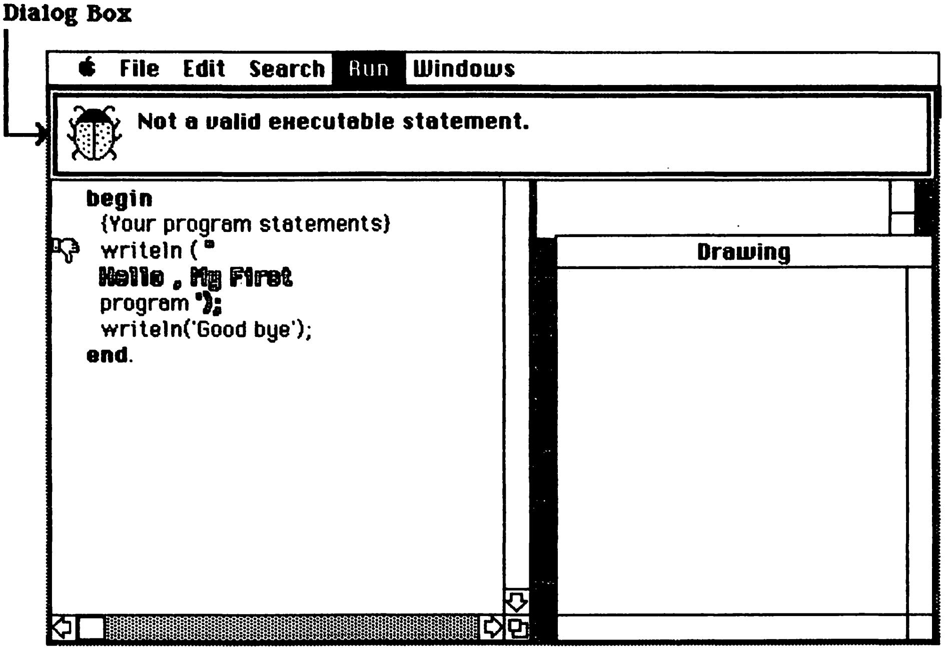
Scrolling a Window

Notice the horizontal and vertical scroll bars in the active window. To make the contents move up or down, click-hold the up or down arrow at either the top or bottom of the vertical scroll bar. Similarly, you can scroll back and forth by click-holding the left or right arrow on either side of the horizontal scroll bar.

Each scroll bar contains a square box indicating what portion of the entire contents of the window has been scrolled at any instant. Click-holding a scroll arrow causes this box to move, but you can also move the box itself in a kind of short-cut method of scrolling. For example, to scroll one half of the way through a document such as a Pascal program, drag the scroll box to the midpoint of the vertical scroll bar.

Dialog, or, Talking Back to Your Macintosh

Once in a while the Macintosh must warn you of some dangerous condition, get information from you to help it out of a fix, or just get a command from you. To do this, a special kind of window called a *dialog box* appears like the one shown in Figure 1.6.



Dialog Box

Not a valid executable statement.

```
begin
  {Your program statements}
  writeln ('
  Hello , My First
  program ');
  writeln('Good bye');
end.
```

FIGURE 1.6
A dialog box showing an error.

Dialog boxes appear when needed and disappear when a crisis is resolved or the Macintosh has the information it needs. Most dialog boxes can be handled using the mouse to click the answer. For example, "bug" dialog boxes are generally informative and therefore do not require responses. Simply click inside the "bug" box to make it disappear before making the necessary correction.

A dialog box will appear each time you save a program on diskette or print a program on the printer, and when you do something Macintosh cannot understand. Some dialog boxes will contain icons, text, or multiple-choice buttons, and some will require you to enter a name, select an item from a list, or simply acknowledge a message by clicking the dialog box. The Macintosh usually "beeps" prior to displaying a dialog box.

Starting Pascal

Suppose we get started using Macintosh Pascal. Insert the Pascal diskette and start your system. Open the Pascal Diskette Window either by pointing to the MacPascal diskette icon (located in the upper right-hand corner of the screen) and double-clicking or by selecting the OPEN option under the FILE menu. Figure 1.7 shows how your screen should look when this is accomplished.

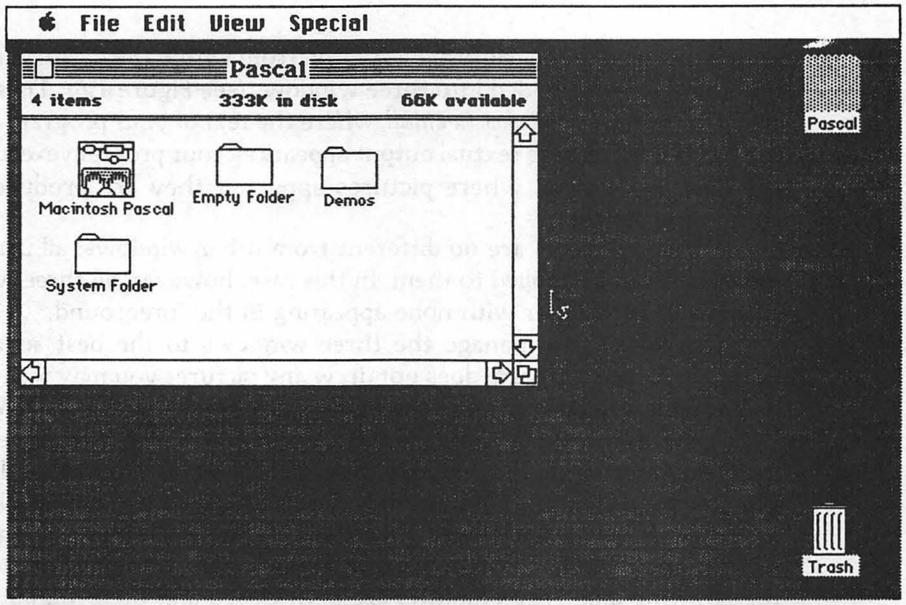
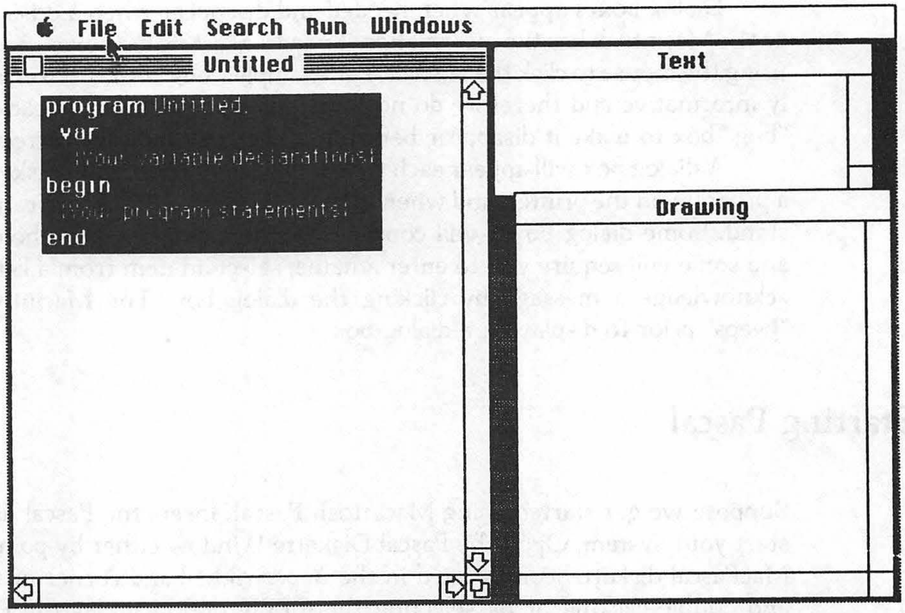


FIGURE 1.7
Pascal disk Open Window.



```
program Untitled;
  var
    {Your variable declarations}
begin
  {Your program statements}
end
```

FIGURE 1.8
Pascal Program, Text, and Drawing Windows.

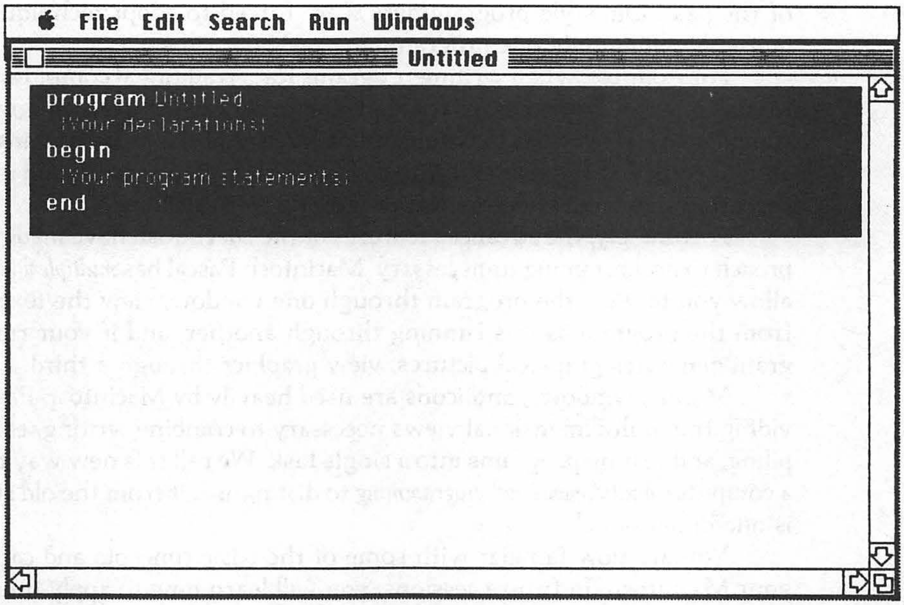
Now open the Macintosh Pascal icon from within the Pascal window. Note that the screen is divided into three windows (see Figure 1.8). These windows are: (1) the *Program Window (untitled)*, where the text of your program appears; (2) the *Text Window*, where textual output appears as your program executes; and (3) the *Drawing Window*, where pictures appear if they are produced by your running program.

These windows are no different from other windows; all usual window operations can be applied to them. In this case, however, all three windows are adjacent to each other with none appearing in the foreground.

The idea is to manage the three windows to the best advantage. For example, if your program does not draw any pictures you may want to remove its Drawing Window and use the extra space to enlarge the Text Window.

How is this done? First, if the Drawing Window is inactive, move the mouse pointer inside the window area and click once. Then close the Drawing Window by placing the pointer on its close box and click one time. Now you can drag the size box in the lower right corner of the Text Window down and to the right, making the Text Window grow larger. Figure 1.9 shows the screen when the Drawing and Text Windows are both closed and the Program Window is enlarged and active.

When entering a new program, you must work in the Program Window. You may choose to leave the Drawing and Text Windows intact or close them,



```
program Untitled;
 {Your declarations}
begin
 {Your program statements}
end.
```

FIGURE 1.9
A Pascal Program Window.

using their space for an enlarged Program Window. Also, all editing (changing) of your program must be done in the Program Window. The Text and Drawing Windows are used when you want to see the program's output.

Quitting Macintosh Pascal

Now let's take a break. Select FILE-QUIT to leave Macintosh Pascal. After a pause, the desk top reappears. Select FILE-EJECT to eject the Macintosh Pascal diskette, and turn the Macintosh off. If you do not eject the diskette, then it can be ejected the next time you turn on the Macintosh by simultaneously pressing and holding the mouse button while turning on the power.

Summary

Traditional programming has dominated computing because of the limitations of computer hardware, but the Macintosh is the first of a line of high-speed, graphics-oriented computers destined to make traditional programming a relic of the past. Old style programmers were forced to adopt techniques that required handling only one programming task at a time.

For example, when writing programs for a traditional computer you must: (1) use a text editor to enter (type in) the program, (2) use a program called a compiler to locate errors in typing or invalid uses of the programming language, and (3) return to the text editor to correct errors. You could spend many hours repeating this process before the program was run a single time.

Fortunately, the advanced features of the Macintosh have made the old approach to programming unnecessary. Macintosh Pascal has *multiple windows* which allow you to view the program through one window, view the textual output from the program as it is running through another, and if your running program generates graphical pictures, view graphics through a third.

Menus, windows, and icons are used heavily by Macintosh Pascal in providing the multidimensional views necessary to combine writing, editing, compiling, and testing programs into a single task. We call this new way to program a computer *multidimensional programming* to distinguish it from the old style, which is one-dimensional.

You are now familiar with some of the basic concepts and capabilities of your Macintosh. In future sessions, you will learn how to apply these features as you work "hands-on" with Macintosh Pascal.

Problem Solving

1. "Mouse around" by doing the following desk top exercises:
   a. Drag a window to a new location on the desk top.
   b. Drag an icon around the desk top.
   c. Change the size of a window.
   d. Pull down the "Apple" Accessories menu and see what is in it.
   e. Open two or more windows and practice making each of them the active window.
2. Open the Macintosh Pascal diskette and double-click the Demos folder. This folder contains sample Pascal programs. Select and double-click one to load it into memory and enter Macintosh Pascal. Next, select RUN-GO and watch what happens to the Pascal windows as the program runs.
3. Open the Macintosh Pascal diskette, select and double-click an icon to get into the Macintosh Pascal system. Close all three windows by clicking each of the three close boxes. Now, how do you suppose these windows can be opened again?
4. Make a list of the operations that can be performed with the mouse. What would you say is the principal advantage of a mouse versus a keyboard?

Session 2:

Entering, Running, and Saving a Pascal Program

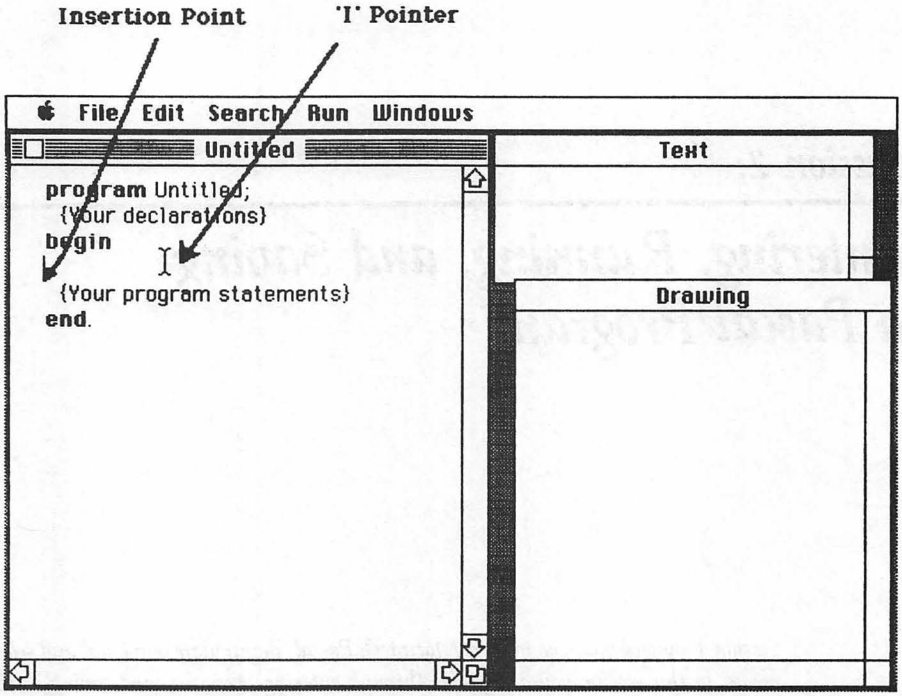
Session 1 showed you how to start Macintosh Pascal, manipulate windows, and select items from a menu. In this session you are guided through entering, running, and saving a simple program.

Running Your First Program

The idea of people using machines for the purpose of writing is not very old. The first typewriter was invented in 1865 by an American named Sholes, and it was perfected by another American, named Remington, in 1877. Even so, the typewriter did not begin to impress many people until the 1880s. Now, 100 years later, the typewriter has been replaced by another writing machine—the personal computer.

For the purposes of this book the Macintosh will become the preferred writing machine for writing Pascal programs. The "Macintosh typewriter" is much more advanced than most ordinary typewriters because it can "remember" text by saving it in diskette files, change text by helping you edit it, and speed up text entry by automatically formatting each line as you enter it. Macintosh typewriters are wonderful, but they require a little practice in order to learn how to use them.

Insert your Macintosh Pascal diskette and start your system. Next open the Macintosh Pascal diskette icon either by double-clicking it or by selecting OPEN from the FILE menu. Finally, open the Macintosh Pascal icon. After a brief pause you should see three familiar windows—*Program (Untitled), Text,* and *Drawing.* See Figure 2.1.



```
program Untitled;
  {Your declarations}
begin

  {Your program statements}
end.
```

FIGURE 2.1
Insertion mark and "I" pointer.

Look closely at the Program Window: it contains the skeleton of a Pascal program in reverse video (white-on-black), which means its contents have been selected. Text is *selected* whenever you want to handle it altogether as a unit (for example, to move or delete it).

If you press the backspace key, the selected contents will be erased. If you begin typing, the keyboard input will replace the selected text. To leave the contents as they appear in the Program Window, simply point and click anywhere within the window.

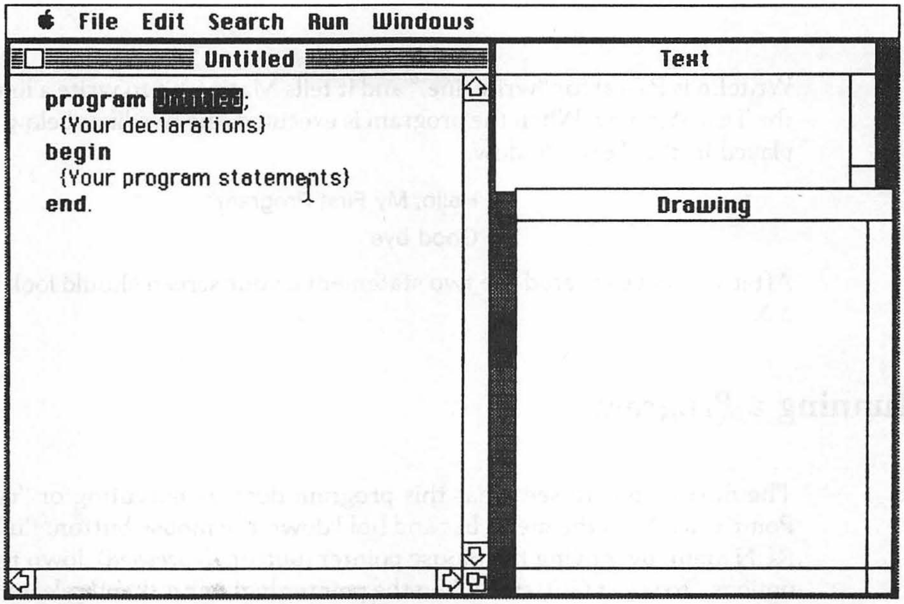
Just for fun (without touching the mouse button and in one sweeping motion), move the pointer starting somewhere on the menu bar, straight down to the scroll bar at the lower edge of the Program Window. Did you notice the pointer changing from an arrow to a large "I" and back again? Inside the Program Window the pointer becomes an "I." By clicking the pointer inside a window you change the "I" into a blinking vertical bar which is called an *insertion point.* The insertion point is where any characters you type will be inserted.

Changing a Line of Program Text

Since the program shown inside the Untitled Program Window is itself untitled, it must be given a name. Move the insertion point to the beginning of the word "Untitled" following the word "Program" in the first line of the Program Window. To replace "Untitled," highlight it by dragging the mouse over it (keeping the button depressed), and then release the button. As you can see in Figure 2.2, the word "Untitled" appears in *reverse* video. From the keyboard, type "Example1" (being careful not to type any spaces), which becomes the name of your first program.

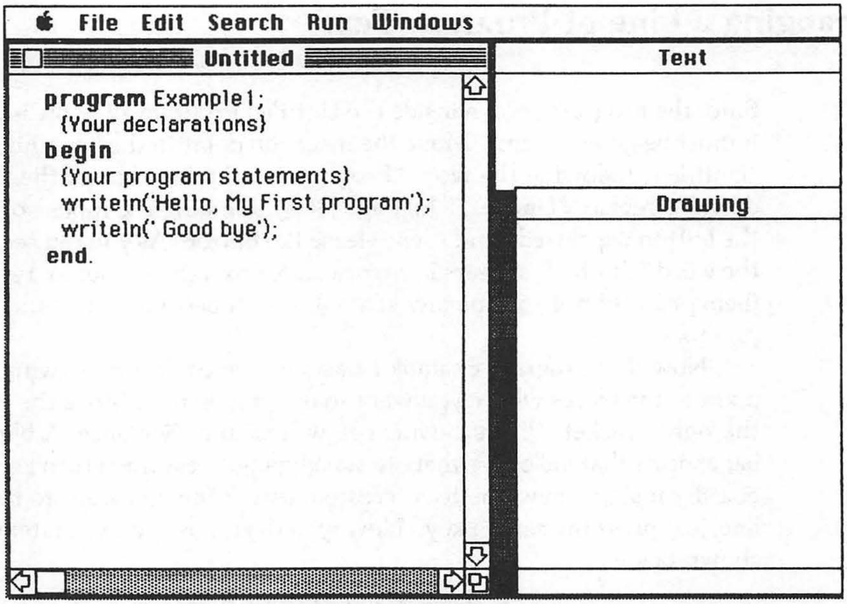
Now that program Example1 has a title, modify it by moving the insertion point to the places where you want to insert new text. Move the cursor beyond the right bracket, "}" on the line below *begin* and click once. A blinking vertical bar appears that indicates where to start typing. Press the return key on your keyboard, causing a new line to be created. (Any time you want to begin on a new line, just press the return key.) Now type the following two statements *exactly* as shown below.

```
WriteLn ('Hello, My First Program') ;
WriteLn ('Good bye') ;
```



```
program Untitled;
  {Your declarations}
begin
  {Your program statements}
end.
```

FIGURE 2.2
Program with "Untitled" shaded.



```
program Example1;
  {Your declarations}
begin
  {Your program statements}
  writeln('Hello, My First program');
  writeln(' Good bye');
end.
```

FIGURE 2.3
Example1 program.

WriteLn is Pascal for "write line," and it tells Macintosh to write a line of text in the Text Window. When the program is executed, the two lines below will be displayed in the Text Window.

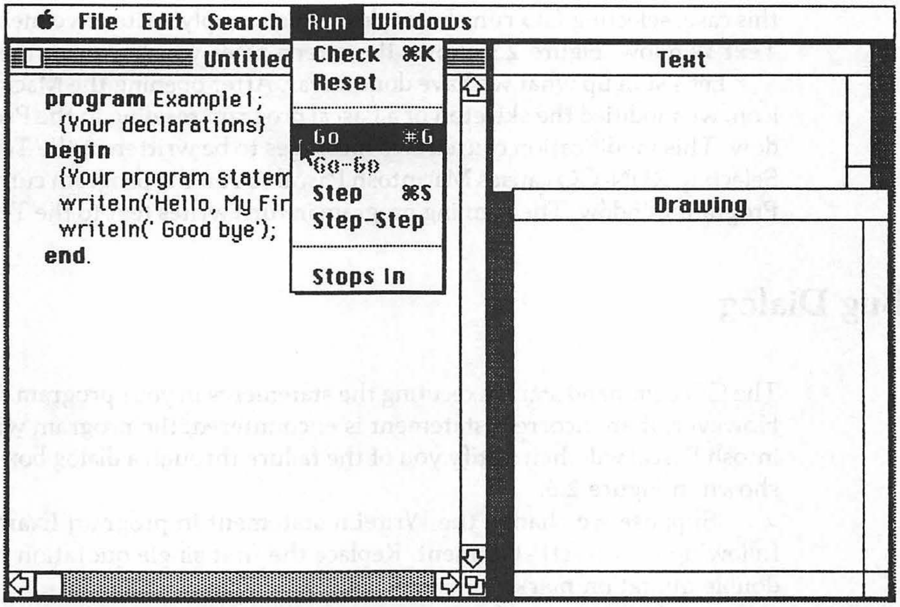
```
Hello, My First Program
Good bye
```

After you have entered the two statements, your screen should look like Figure 2.3.

Running a Program

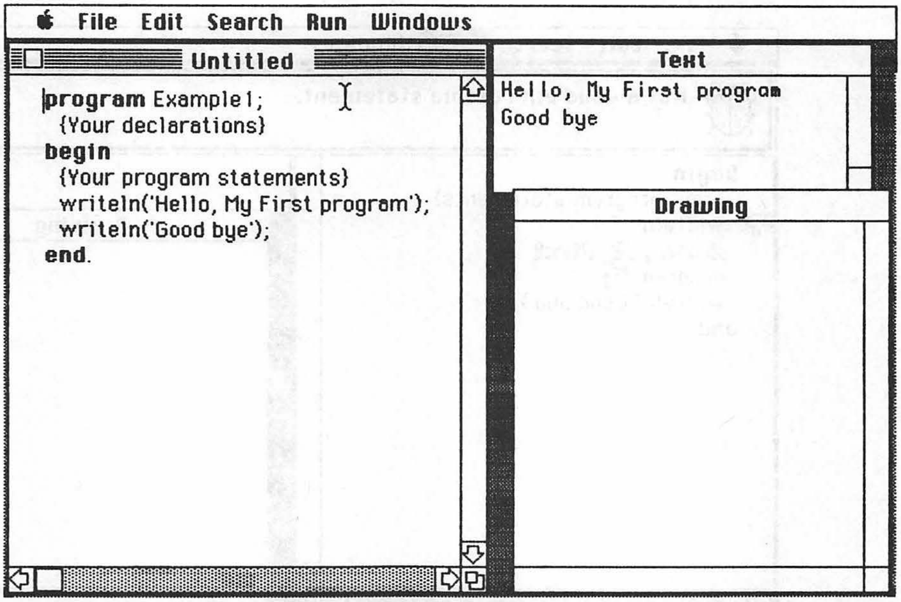
The next step is to see what this program does by executing or "running" it. Point at RUN on the menu bar and hold down the mouse button. Pull down the RUN menu by moving the mouse pointer (button depressed) down through the options. To select GO, stop when the pointer highlights it and release the mouse button. Watch what appears in the Text Window. Figure 2.4 shows the screen before you release the mouse button while selecting Go.

As you can see, there are many options to choose from within the RUN menu. For the time being, GO is all you need. Selecting GO informs Macintosh Pascal that you want to run the program currently in the Program Window. In



```
program Example1;
  {Your declarations}
begin
  {Your program statem
  writeln('Hello, My Fir
  writeln(' Good bye');
end.
```

FIGURE 2.4
Use of "Go" to Run Example1.



```
program Example1;
  {Your declarations}
begin
  {Your program statements}
  writeln('Hello, My First program');
  writeln('Good bye');
end.
```

```
Hello, My First program
Good bye
```

FIGURE 2.5
Results of "Go" on Example1.

this case, selecting GO runs Example1, which simply writes two messages in the Text Window. Figure 2.5 shows the screen after you have run the program.

Let's sum up what we have done so far. After opening the Macintosh Pascal icon, we modified the skeleton of a Pascal program residing in the Program Window. This modification caused two messages to be written in the Text Window. Selecting RUN-GO causes Macintosh Pascal to run the program currently in the Program Window. The running program in turn writes text to the Text Window.

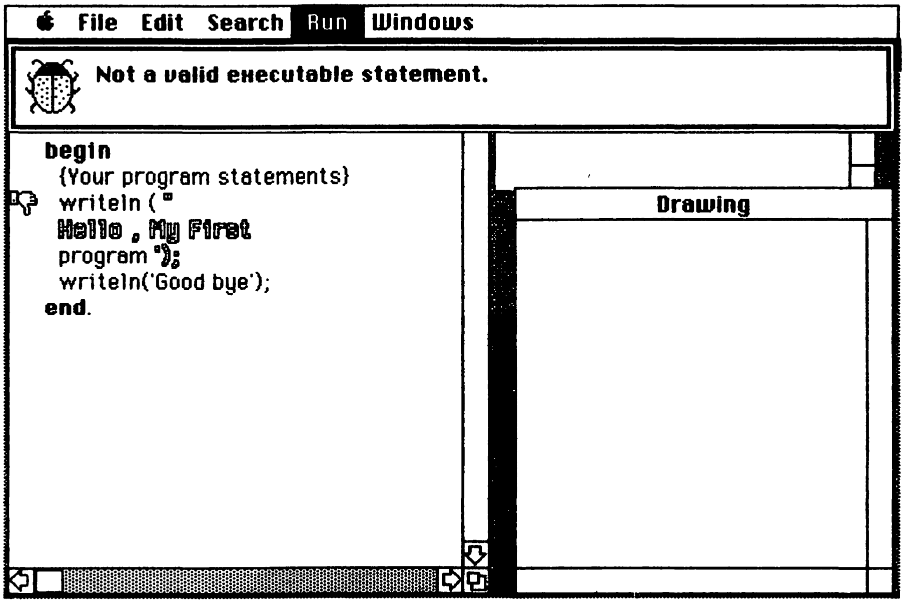
Bug Dialog

The GO command starts executing the statements in your program sequentially. However, if an incorrect statement is encountered, the program will fail. Macintosh Pascal will then notify you of the failure through a dialog box like the one shown in Figure 2.6.

Suppose we change the WriteLn statement in program Example1 to the following (incorrect) statement. Replace the first single quotation mark with a double quotation mark.

```
WriteLn ("Hello, My First Program') ;
```

If you don't remember how to make this change, follow these directions. First move the cursor to the right of the left parenthesis and drag the mouse over the



Not a valid executable statement.

```
begin
 {Your program statements}
 writeln ("
 Hello , My First
 program ");
 writeln('Good bye');
end.
```

FIGURE 2.6
Example of BUG message dialog box.

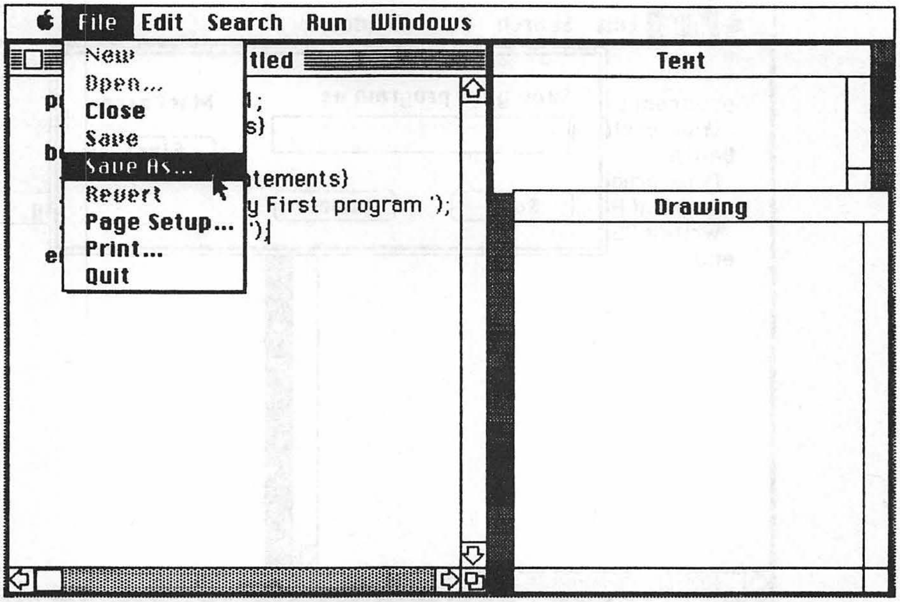
single quotation mark. Type a double quotation mark in place of the highlighted single quotation mark.

The only change you made was to replace a single quote with a double quote. Try to run this program by selecting GO from the RUN menu. This time the program produces a surprising result—A BUG message appears informing you of some kind of error. (A common synonym for a programming error is the word "bug.") In addition, notice that the vicinity in which the error occurred is shown in outline style and is flagged with a "thumbs-down" symbol. Macintosh Pascal is saying that the thumbs-down line is not a correct Pascal statement; hence it is unable to execute it.

To get rid of the bug, make the message disappear by moving the pointer anywhere in the error message window and clicking once. Using the same method as before, delete the offending double quote and insert a single quote in its place. Make sure your program is bug-free by running it again, successfully.

Saving Your Program for Later Use

Programs in the Program Window should be saved to diskette so you can use them later. Never attempt to save programs on the Macintosh Pascal diskette, which ought to be *write protected* so it cannot be written on. Instead, use another diskette to keep your programs on.



```
p              l;
               s}
b
               tements}
               y First program ');
               ')
e
```

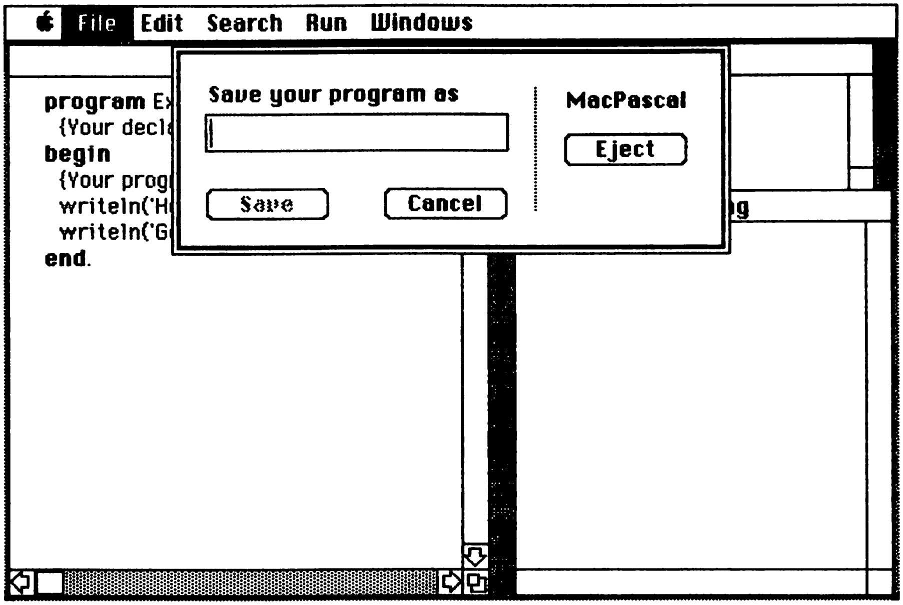
FIGURE 2.7
File option to Save a new program.

To save a program, first select the menu item SAVE AS . . . from the items under FILE. This is done by pulling down the FILE menu and selecting SAVE AS . . . (see Figure 2.7). Shortly, a dialog box will appear asking for the name under which you wish to save your program. Remember, dialog boxes either inform you of a crisis (recall the error dialog box with the bug icon you saw earlier) or notify you of options and wait for your response.

In this example, you are given four alternatives to choose from (see Figure 2.8). The first is a blank box on which to enter the name of the file you want to save your program under. The three remaining options are enclosed in labeled boxes called *buttons:* SAVE (dimmed), CANCEL (highlighted), and EJECT (highlighted).

Why are some of these dialog buttons highlighted and some dimmed? Since there are only two logical courses of action to choose from at this decision point, there are only two corresponding operative (highlighted) buttons. Selecting the CANCEL button allows you to change your mind and not save the program. Choosing CANCEL returns you to where you were before you selected SAVE AS . . . from the FILE menu.

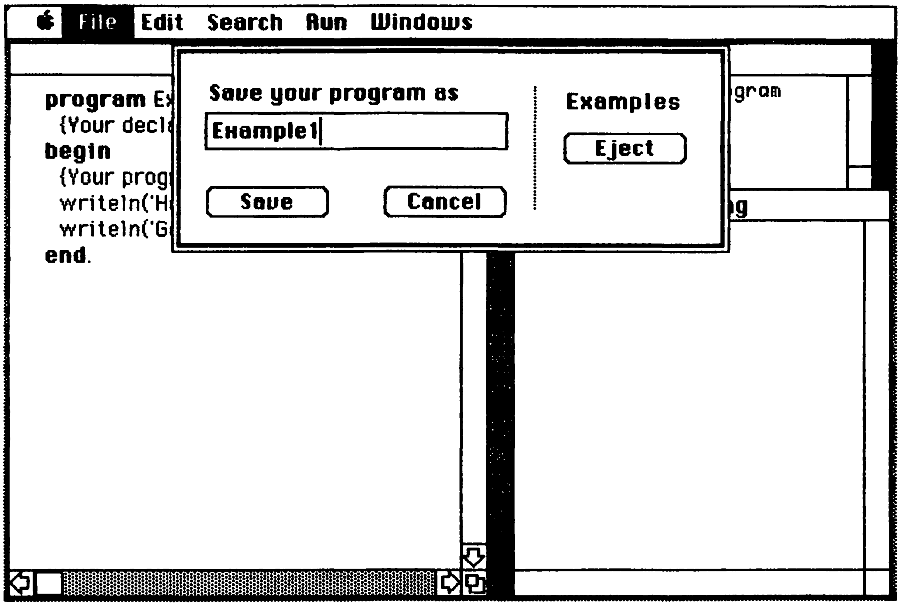
In Macintosh terminology, the highlighted buttons (CANCEL and EJECT) are said to be "enabled," that is, they are ready and able to respond to your commands. Conversely, the dimmed button (SAVE) is "disabled" or not able to carry out its action until such time as it is again enabled. If you select SAVE, for example, prior to entering a name for your program in the name box, no action will be taken because SAVE is disabled.



```
program E
  {Your decl
begin
  {Your prog
  writeln('H
  writeln('G
end.
```

Save your program as

FIGURE 2.8
Dialog box for selection of Save As. . .



```
program E
  {Your decl
begin
  {Your prog
  writeln('H
  writeln('G
end.
```

Save your program as

Example1

FIGURE 2.9
Save dialog after program name entered.

A different action is taken if you wish to save the program on another diskette. In this case, EJECT the Macintosh Pascal diskette. Dialog boxes will appear which guide you through the process. Note that the name of the diskette currently in the drive always appears above the EJECT button so that you can readily identify which diskette is in the drive.

After the Pascal diskette is ejected from the drive, the EJECT button is disabled since you cannot eject a diskette from an empty drive. Remove the ejected Pascal diskette and insert an alternate diskette to save the program on. This is a good time to enter the name to save your program under, so type Example1 in the name box. As soon as you start typing, SAVE becomes enabled. (See Figure 2.9.)

Select SAVE and your program will be written on the alternate diskette and saved for future use. When you open this diskette window later, a program icon named Example1 will appear as shown in Figure 2.13.

After the program has been saved, the alternate diskette is ejected and a new dialog box asks you to insert the Macintosh Pascal diskette. After you do so, you are returned to what you were doing before you selected SAVE AS. . . .

Meanwhile, back at the Program Window, notice how its title (the filename) has changed from "Untitled" to "Example1" because you named and saved the program. (See Figure 2.10.)

Also notice that the file name and program name are identical. It is a good idea to save programs in files of the same name to reduce confusion. If you're entering a long program, save what you've entered several times before you reach the end. If a power shortage or other disaster occurs and your program is lost, you still have the saved version and can avoid having to retype the entire program.

Editing

There are times when you need to add, delete, or modify your program and then save the changed version. To *delete* a portion of your program, highlight the area to be omitted by dragging the mouse over it (button depressed), release the button, and press Backspace on the keyboard, once.

To make additions to a program, move the insertion point to the desired place and type. For example, program Example1 can be changed by editing the following lines.

1. Add these two lines after "{your declarations}":

```
Var
   count: integer;
```

2. After "WriteLn ('Hello, My first Program');" add:

```
For counter = 1 To 50 Do
WriteLn ('counter = ' ,counter) ;
```

The Program Window should look like Figure 2.11.

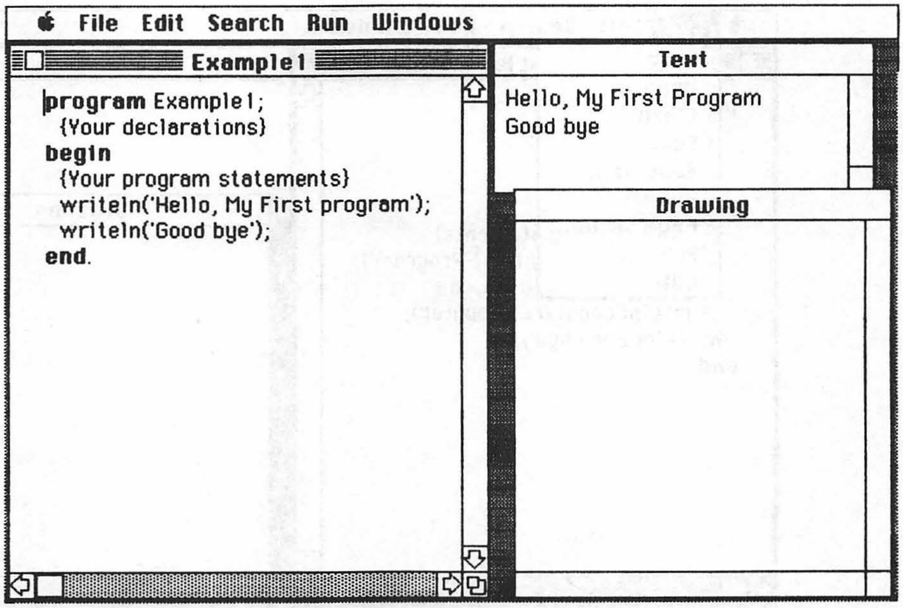
(Optional: If you are curious to know what effect these additions have made, select GO from the RUN menu and watch the Text Window.)

If you want to modify a word or group of words, highlight them using the mouse and just start typing in the change. The highlighted text will be replaced immediately by whatever you enter.

Saving Your Edited Program

If you pull down the FILE menu after saving Example1, more buttons are enabled than before it was saved. Obviously, a program must be saved before the system will let you perform these other operations on it. Compare Figures 2.12 and 2.7.

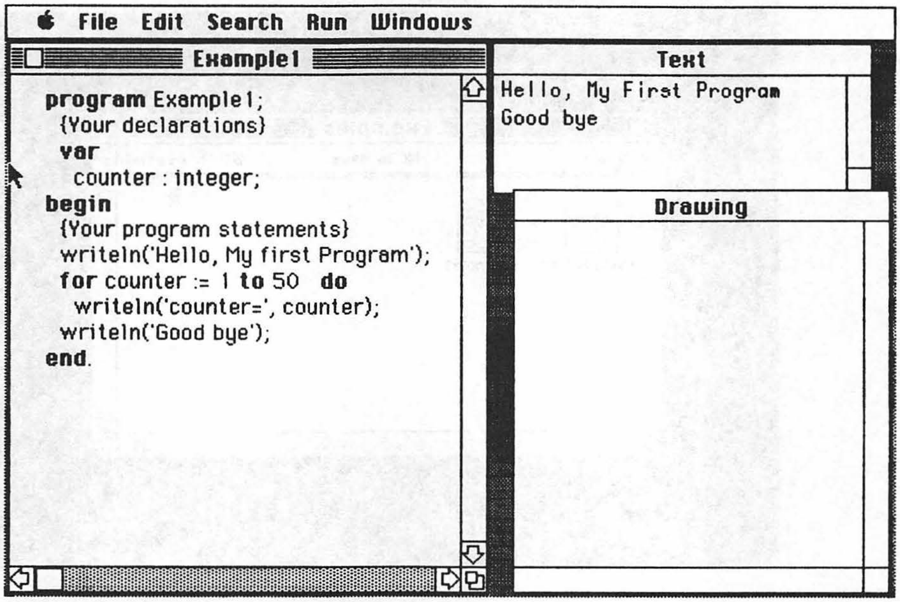
You can easily replace a version of Example1 saved on diskette with a changed version in the Program Window by selecting FILE-SAVE. When you do, no dialog box appears to request a program name because Macintosh Pascal recalls that you've saved this once before and which diskette it was saved on. A dialog box does appear asking for the proper diskette (by name) to be inserted into the drive. You are again guided through the saving process, and then returned to the Pascal Program Window, as before.



```
program Example1;
  {Your declarations}
begin
  {Your program statements}
  writeln('Hello, My First program');
  writeln('Good bye');
end.
```

```
Hello, My First Program
Good bye
```

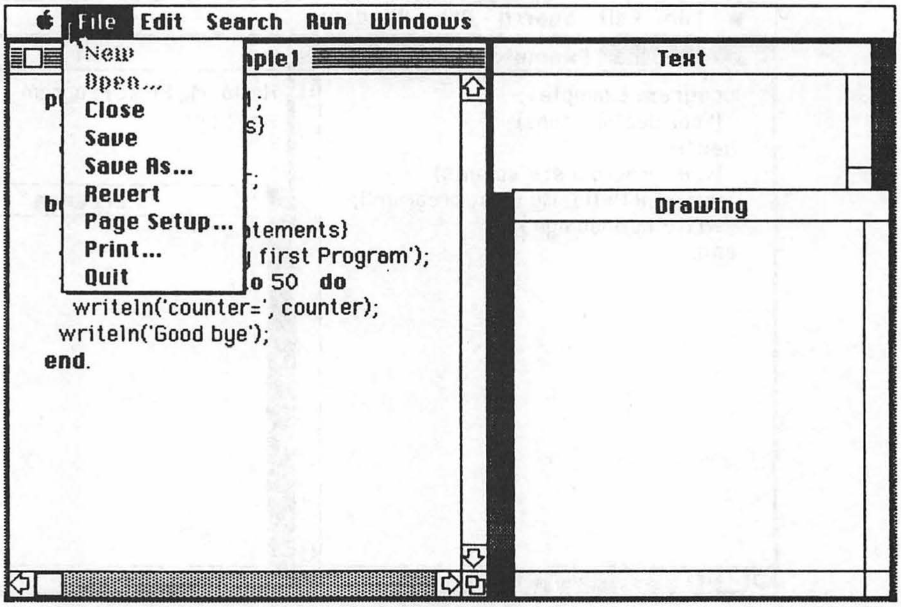
FIGURE 2.10
Effect of Saving on Program Window title.



```
program Example1;
  {Your declarations}
  var
    counter : integer;
begin
  {Your program statements}
  writeln('Hello, My first Program');
  for counter := 1 to 50  do
    writeln('counter=', counter);
  writeln('Good bye');
end.
```

```
Hello, My First Program
Good bye
```

FIGURE 2.11
Example1 after modification.



```
p                ;
                 s}

                 ;
b
                 tements}
                 first Program');
                 o 50  do
    writeln('counter=', counter);
  writeln('Good bye');
end.
```

FIGURE 2.12
File menu, more options after program is saved.

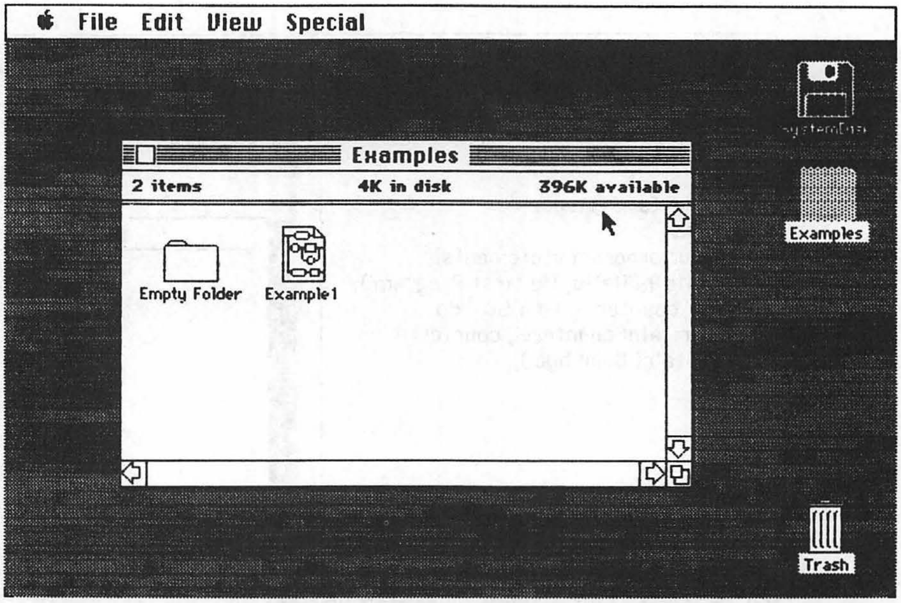


FIGURE 2.13
Example1 icon.

Alternately, if you wanted to save the changed program under a new name (which doesn't affect the program saved under the old name) select FILE-SAVE AS . . . and supply the new name. To terminate this session, select FILE-QUIT.

Restoring a Saved Program

To copy a previously saved program from a diskette file into main memory, you must first FILE-CLOSE the existing program and then FILE-OPEN another program.

Pull down the FILE menu and select the CLOSE item. If the current program has not been saved, you will be asked if you want to save or discard it (a dialog box appears with SAVE, DISCARD, and CANCEL buttons). However, if the current program has been saved (and not modified since being saved) then another dialog box will appear with an EJECT button.

Click the EJECT button and replace the Macintosh Pascal diskette with the diskette containing the program you want to copy into memory. As soon as the diskette is loaded, the dialog box will display the files on the diskette in a small window which can be scrolled.

Find the program file you want and double-click it. This will cause the program to be loaded into main memory, and a dialog box will appear telling you to put the Pascal diskette back into the drive.

The newly loaded program will appear in the Program Window. You can edit, run, or save this program as before.

Printing the Current Program

To print the program currently in the Program Window, you must first make sure the write protect tab on your Pascal diskette is closed (the write protect is *off*) and your printer is turned on (plugged in, loaded with paper, etc.).

Select the FILE-PRINT item and wait for the printer-setup dialogue box to appear. This dialog box has several buttons for choosing quality of print and page dimensions. Any of these settings will work, and in most cases you will simply click the OK button to continue.

Macintosh Pascal will write the image to be printed to disk first, then print the disk file. To terminate before the entire file is printed, simultaneously press the APPLE control key (located immediately to the left of your keyboard spacebar) and the period key. Otherwise, sit back and relax while the printer gives you a hard copy of your Pascal program.

When you are finished, eject the diskette and open the write protect tab (turning *on* the write protect) so you cannot accidentally erase essential files.

Summary

You will be using two diskettes while learning Macintosh Pascal. The Macintosh Pascal diskette contains the Pascal interpreter software and should be *write protected* so you cannot accidentally erase the interpreter. The other diskette contains your own programs.

Programs are entered, edited, run, and printed from the Program Window. To keep a permanent copy of each program, FILE-SAVE AS . . . or FILE-SAVE the contents of the Program Window onto the second diskette. Use FILE-SAVE AS . . . the first time a program is saved or whenever a new name is desired, and FILE-SAVE whenever the Program Window already knows the file name.

An old program on diskette is loaded into the Program Window by the FILE-OPEN command. If the program is on another diskette, EJECT the Macintosh Pascal diskette and follow the directions given in the dialog boxes. This process is similar to FILE-SAVE

A program is executed by selecting RUN-GO. Its output appears in the Text Window. The RUN menu has several other items for running a program. These options will be discussed when the need arises, but for now use RUN-GO to execute a program.

Problem Solving

1. Use FILE-OPEN to copy Example1 from your second diskette into the Program Window. List the steps needed to do this.
2. How are the mouse and keyboard used to delete an entire line of text from the Program Window?
3. Perform the following experiment. Close the Drawing Window and stretch the Program Window so that it nearly conceals the Text Window (but do not conceal it entirely). How is the Text Window brought in front of the Program Window without re-sizing either window?
4. What happens when you attempt to select a dimmed item from a menu or a dimmed button from a dialog box?
5. Explain how to insert a line of text between two existing lines in the Program Window.

Session 3:

The Structure of a Pascal Program

The text that first appears in the Program Window of Macintosh Pascal (immediately after opening the Pascal icon) is the smallest possible complete Pascal program. Although it does nothing meaningful, it is nevertheless a valid program. In this session, you will learn about the structure of data, instructions, constants, and variables in Pascal. This will enable you to write meaningful programs that carry out calculations and do simple input and output operations.

Reserved Keywords

Start up Macintosh Pascal and look at the skeleton program shown highlighted in the Program Window. Click the Program Window to un-select "Program (Untitled)."

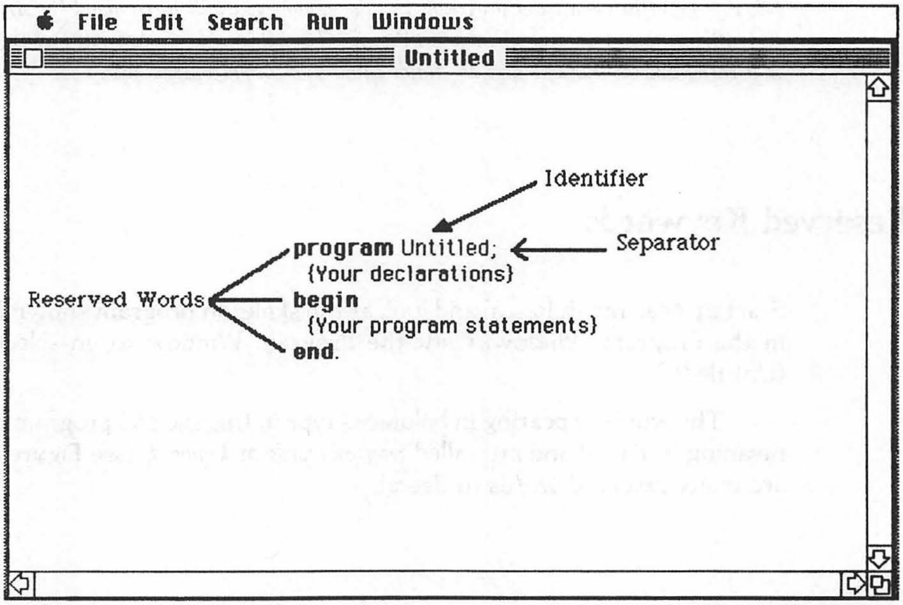
The words appearing in boldfaced type in this skeletal program have special meaning in Pascal and are called *reserved words* or *keywords* (see Figure 3.1). There are many reserved words in Pascal:

AND
ARRAY
BEGIN
CASE
CONST
DIV
DO
DOWNTO
ELSE
END
FILE
FOR
FUNCTION
GOTO
IF
IN
LABEL
MOD
NIL
NOT
OF
OR
OTHERWISE
PACKED
PROCEDURE
PROGRAM
RECORD
REPEAT
SET
STRING
THEN
TO
TYPE
UNTIL
USES
VAR
WHILE
WITH

"Reserved" means that these words cannot be used in any way except as punctuation symbols in the grammar of a Pascal program. The rules of grammar are called syntax rules, and in Pascal these rules are defined by *syntax diagrams.*

Syntax Diagrams

You can use a syntax diagram like the simple one shown in Figure 3.2 to check the precise grammar of any Pascal statement. Following the diagram in the direc-



```
program Untitled;
 {Your declarations}
begin
 {Your program statements}
end.
```

FIGURE 3.1
A complete program skeleton.

tion of the arrows (left to right), shows you how words are put together to form valid Pascal statements.

In a syntax diagram circles always appear around reserved words and special symbols (such as the ; or .). Circled words appear in a Pascal statement just as they are shown inside the circle. Boxes are used to enclose other syntactic forms. Each separate syntactic form has its own diagram; for example, there are diagrams which define the syntax of an identifier, a declaration part, and the body of a program.

Look at Appendix B, "Pascal Syntax Diagrams," in the back of this book to see the complete set of syntax diagrams for the Pascal language. By learning to read and understand these diagrams, you will be able to verify correct usage of statements within a Pascal program.

Program Statement

The first line of a Pascal program is called the *program statement* (or the "header") and always starts with the reserved word *"Program"* and ends with a semicolon. (All statements in a Pascal program must end with a semicolon except the statements immediately preceding the reserved words *"END," "ELSE,"* and *"UNTIL.")* The word following the reserved word "Program" is the name or title given to the program by the programmer. The usual term for this program title is the program *identifier.*

Identifiers

Identifiers are not limited to identifying programs only. They are used for naming a variety of objects in Pascal. An *object* can be a number, a place in memory where a number is stored, a program, or even a series of actions within a program.

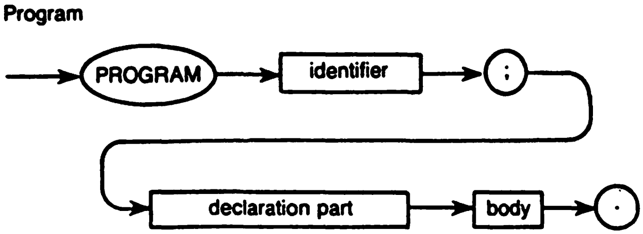


FIGURE 3.2
Example of a syntax diagram.

Certain rules must be observed while creating a Pascal identifier: (1) it must begin with a letter, (2) the subsequent characters must be strings of letters or digits, and (3) it must be fewer than 255 characters in length.

The rule for an identifier excludes punctuation marks, spaces, or special characters. (Exception: you may use the underscore within an identifier to enhance readability, for instance, My_Program_Draws_Circles is acceptable.) The following identifiers are legal:

```
FIVE
MyFirstProgram
M2D2
Counts_Capital_Letters
```

while these are not:

```
2daysProgram
_Break
Bar.Graph?
$MoneyMaker
Pie Graph
```

Program Body

The program body comes after the program header and is where program instructions are listed. Its beginning is marked by "{Your Program Statements}" in the skeleton program. Note that statements appearing in the program body are enclosed by the reserved words "Begin" and "End." (Refer back to Figure 3.1.)

Declaration Part

The program body is preceded by a section called the *declaration part,* which defines all objects used by the program body. In the skeleton program the declaration part is marked with the comment "{Your Declarations}".

A simple declaration part contains two statements (in order of appearance): (1) the *constant definition* statement, followed by (2) the *variable declaration* statement. Only legal Pascal identifiers may be used as variable or constant names. Other statements may appear in the declaration part as well, but for now let's see how these two simple statements work.

Types

All objects of Pascal must be typed. A *type* is a collection of values. For example, the set of whole numbers -32,768, -32,767, . . . 0,1,2, . . . 32,767 is a collection of values called *integers* in Pascal. An integer object contains integers *only*, and so we say its type is *Integer.*

The set of positive and negative numbers that contain a decimal point is called the *Real* type. A real number such as 5.2 must be stored as a real object.

Another type, called *Boolean*, holds *only* the logical values True and False.

A character object contains only letters, digits, special characters, and whatever can be entered by a single keystroke; its type is *Char.*

The simplest types in Pascal are Char, Real, Integer, and Boolean; therefore, the simplest objects in Pascal must be either Char, Real, Integer, or Boolean. But there are many more types possible in Pascal. For a start, suppose we examine only the simplest objects. There are two ways to declare simple objects: constants or variables.

Constants

Constants are defined as objects which cannot be altered or changed by any subsequent action of the program. Once a constant is defined, it can be used throughout the program in place of the value it represents.

The type of data a constant identifies depends on the value associated with it. It could be a Real type (if it is a number containing a decimal point), an Integer type (if it is a whole number without a decimal point), or a Char type (if it is a single character or special character).

The reserved word "Const" is written at the beginning of the constant definition statement. The constant's identifier, an equals sign, and its value follow. Successive constant identifiers and their values can be listed, separated by semicolons. The following constant definition statement defines three constants: FIVE (Integer), PI (Real), and Rating (Char).

```
Program Sample;
{Your declarations}

Const
        FIVE    = 5:
        PI      = 3.14;
        RATING = '*';

Begin
{Your program statements}
WriteLn ('This is a  ',FIVE,RATING) ;
WriteLn ('This value of PI is ' ,PI:10.2)
End.
```

Try this program by inserting the Const statement and the WriteLn statements into the skeleton program and then selecting RUN-GO.

Look in the Text Window after Program Sample has been run. The quoted messages inside the two WriteLn statements are shown followed by the value of FIVE, RATING, and PI. In the second WriteLn statement, PI was displayed in a format given by :10:2. This means to allow 10 columns in the Text Window for the entire number with 2 of the 10 columns devoted to the decimal fraction of PI's numerical value (3.14).

FIVE is an Integer constant because it is given an integer value (5 has no decimal point). RATING is a Char constant because it is given a '*' as its value. PI is a Real constant because it contains a number with a decimal point.

Variables

Variables are objects which store information of a certain type. There is, however, a critical difference between a variable and a constant: variable values can change or be changed (vary) during program execution. Although a variable contains a value of a certain data type (like a constant), its type must be specified by name (not implied as in the Const statement).

The reserved word "*Var*" is used to begin the variable declaration statement. Var is followed by each declaration consisting of the variable identifier, a colon, and a data type. Successive variable declarations are separated by semicolons.

```
Var
     CandyJar    : Integer ;
     GPA         : Real ;
     LetterGrade : Char ;
```

Several variables of the same type can be listed together but separated by commas as shown below.

```
Var
     X,Y,Z       : Integer ;
     A,B,C:      : Real ;
```

The Var statement is optional, but if it appears in a program, it must follow the Const statement (or the program header if there is no Const). All variables must be declared in a single Var statement.

Assigning a Value to a Variable

Suppose we create a variable named CandyJar of type Integer, in order to count pieces of candy (data) stored in a hypothetical candy jar. CandyJar should be considered initially empty after it has been defined in the Var statement.

A value of 1 can be stored in CandyJar using a Pascal *assignment* statement in the program body, as follows:

```
CandyJar := 1 ;
```

The "assignment" operator (:=) causes the value to its right (1) to be copied into the variable to its left (CandyJar).

If you want to copy the contents of one jar and place them in another, use the assignment statement to copy from right to left.

```
CandyJar := OldCandyJar;
```

This assignment statement copies from the contents of OldCandyJar to the contents of CandyJar. The contents of OldCandyJar are not changed, but the previous contents of CandyJar are replaced by the copy of OldCandyJar's contents.

Suppose there are three candies in the CandyJar; you could be greedy and add two more candies to it. This is accomplished by the assignment statement:

```
CandyJar := CandyJar + 2;
   (5) <------ (3) + (2)
```

Pascal takes the contents of the CandyJar (to the right equalling 3), adds two more to it (totalling 5), and places a copy of that total in the most current CandyJar (to the left of the assignment operator).

Hands on the CandyJar

Let's write a program to illustrate some of these new programming concepts. After you've started up Macintosh Pascal, follow these instructions:

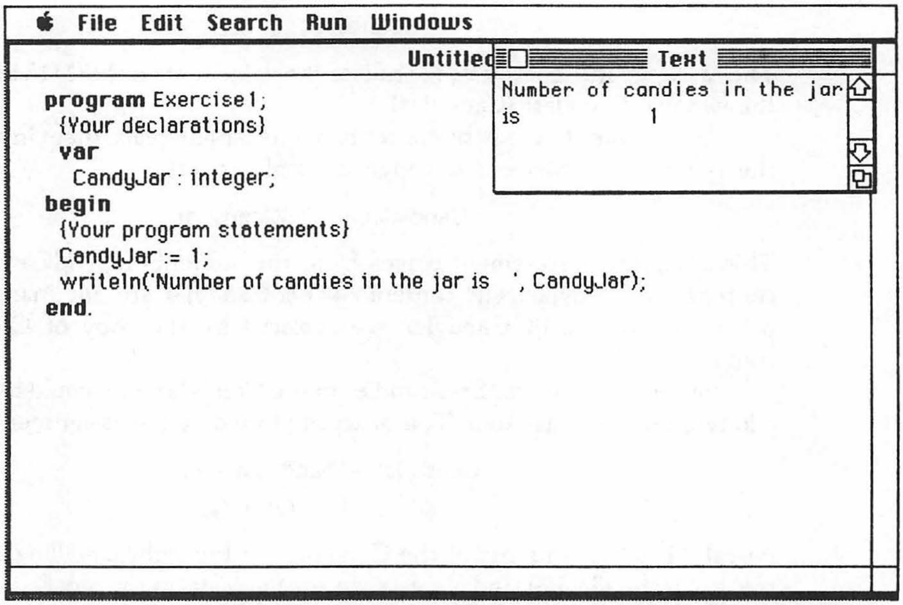
1. Close the Text and Drawing Windows. Then enlarge the Program Window as much as possible (stretch it by dragging the size box).
2. Replace "Untitled" in the program statement with the identifier EXERCISE1.
3. Enter a variable declaration statement as follows:

```
VAR
      CandyJar : Integer ;
```

4. Enter the following Program statements to the program body between Begin and End.

```
CandyJar := 1;
WriteLn ('Number of candies in the Jar is ' ,CandyJar) ;
```

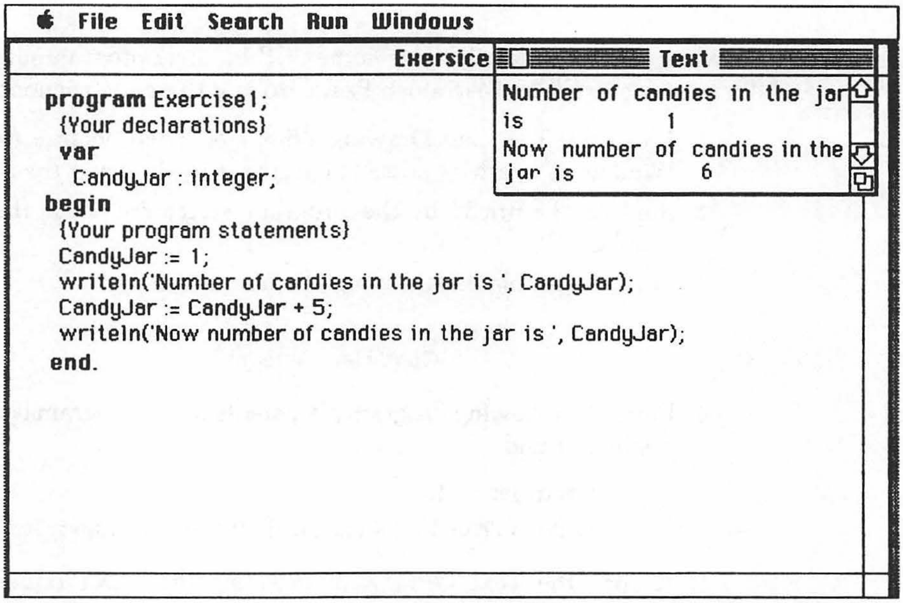
5. Reopen the Text Window by selecting the TEXT option from the WINDOWS menu.
6. Run the program and watch its output in the Text Window. See Figure 3.3.



```
program Exercise1;
  {Your declarations}
  var
    CandyJar : integer;
begin
  {Your program statements}
  CandyJar := 1;
  writeln('Number of candies in the jar is   ', CandyJar);
end.
```

```
Number of candies in the jar
is           1
```

FIGURE 3.3
Exercise1 assigning 1 to CandyJar.



```
program Exercise1;
  {Your declarations}
  var
    CandyJar : integer;
begin
  {Your program statements}
  CandyJar := 1;
  writeln('Number of candies in the jar is ', CandyJar);
  CandyJar := CandyJar + 5;
  writeln('Now number of candies in the jar is ', CandyJar);
end.
```

```
Number of candies in the jar
is           1
Now number of candies in the
jar is          6
```

FIGURE 3.4
Exercise1 after adding new changes.

7. Just to be greedy, add the following statements after your last WriteLn (see Figure 3.4).

```
CandyJar := CandyJar + 5;
WriteLn ('Now number of candies in the Jar is ' ,CandyJar) ;
```

8. Run Exercise1 again (without drooling). See Figure 3.4.
9. Save this program on another diskette as Exercise1. You will need it again later in this session.

Communicating with the Outside World

Now that you have run a few sample programs it is time to learn more about input and output. The simplest form of *input* comes from the keyboard. This is controlled by the *ReadLn* statement. Similarly, the simplest form of *output* is to the Text Window using *WriteLn*.

WriteLn vs. Write

In every WriteLn statement the output is always enclosed in parentheses. Whatever falls within the single quotes is printed exactly as it is written, for instance:

```
Writein ('He called her cat a furry peripheral');
Writein ('Her cat called him a prr . . t meow') ;
```

Each WriteLn prints its line of text on a separate line. Write is similar to WriteLn except it does not force a carriage return following its output. (Wherever we talk about a "return" we mean the character generated by pressing the Return key on the Macintosh keyboard. This has the effect of moving the cursor to the beginning of the next line.) The following example illustrates by comparison the difference between WriteLn and Write.

```
Begin
   Write  ('Enter X') ;
   WriteLn  ('OK') ;
   WriteLn  ('Enter Y') ;
   WriteLn  ('OK, Too')
End.
```

The output appears in the Text Window exactly as shown below.

```
Enter XOK
Enter Y
OK, Too
```

The first Write statement causes 'Enter X' to be output without a Return. The next statement displays 'OK' immediately following 'Enter X' because no Return has been output to separate the two.

Compare the first line of output with the output displayed by the next two statements. 'Enter Y' is output followed by a Return. Thus when 'OK, Too' is displayed, it appears on the next line.

The WriteLn statement:

```
WriteLn ('OK') ;
```

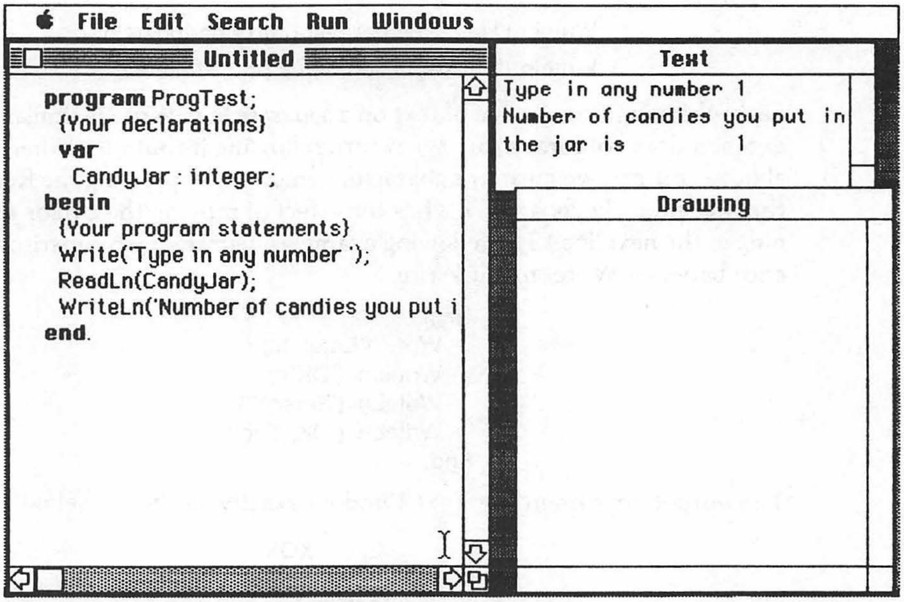
is equivalent to the following:

```
Write('OK') ;
WriteLn;
```

The WriteLn without parentheses simply causes a Return to be output to the Text Window.

ReadLn

What if you want to communicate with or provide input to a running program? You can assign a variable its value from outside the program using the *ReadLn* (pronounced "read line") statement. To see how this works open the Pascal Program Window and enter a program named ProgTest with variable CandyJar of type Integer, as you did in Exercise1. (See Figure 3.5.) Then enter these statements in the Program Body:



```
program ProgTest;
 {Your declarations}
var
  CandyJar : integer;
begin
 {Your program statements}
 Write('Type in any number ');
 ReadLn(CandyJar);
 WriteLn('Number of candies you put i
end.
```

```
Type in any number 1
Number of candies you put in
the jar is           1
```

FIGURE 3.5
ProgTest, use of ReadLn.

```
Write ('Type in any number ') ;
ReadLn (CandyJar) ;
WriteLn ('Number of candies you put in the jar is ', CandyJar) ;
```

Run the program, and when you see the insertion point appear after the line requesting a number, enter any number and hit the Return key. (If you enter more than one value after the prompt, all but the first value will be ignored.) You can see by the final output line that you have successfully assigned a value to variable CandyJar.

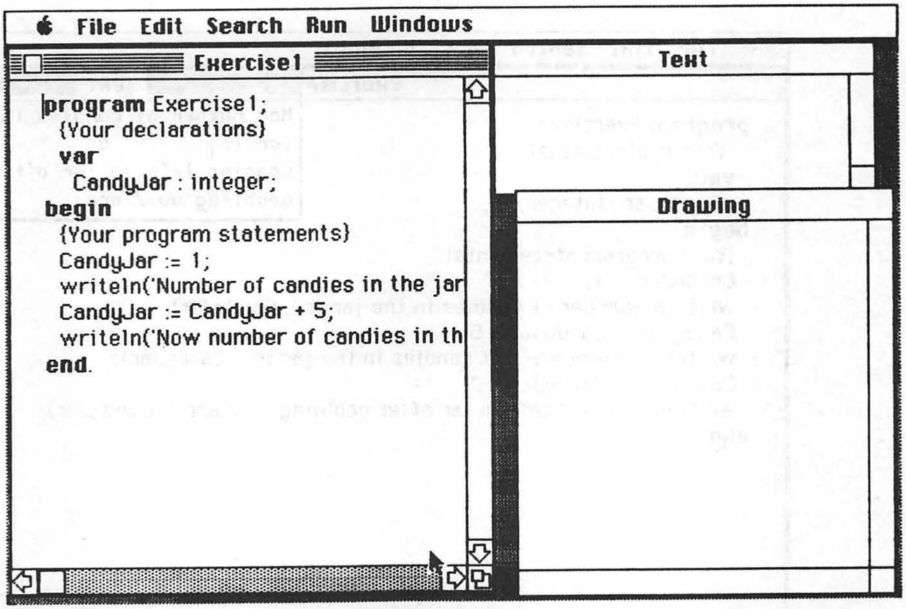
As shown, ReadLn is followed by the name of a variable enclosed in parentheses.

```
ReadLn (CandyJar) ;
```

An insertion point prompt will appear whether or not a polite request generated by a WriteLn statement is present before the ReadLn.

Assigning values from the keyboard using ReadLn makes this program an *interactive* program, because you interact with it. You can assign values to many variables in a running program by listing their names in ReadLn. For example, values for variables X,Y,Z, and LAST can be obtained through a simple ReadLn statement.

```
ReadLn (X,Y,Z,LAST) ;
```



```
program Exercise1;
  {Your declarations}
  var
    CandyJar : integer;
begin
  {Your program statements}
  CandyJar := 1;
  writeln('Number of candies in the jar
  CandyJar := CandyJar + 5;
  writeln('Now number of candies in th
end.
```

FIGURE 3.6
Exercise1 after Open.

A Complete Example

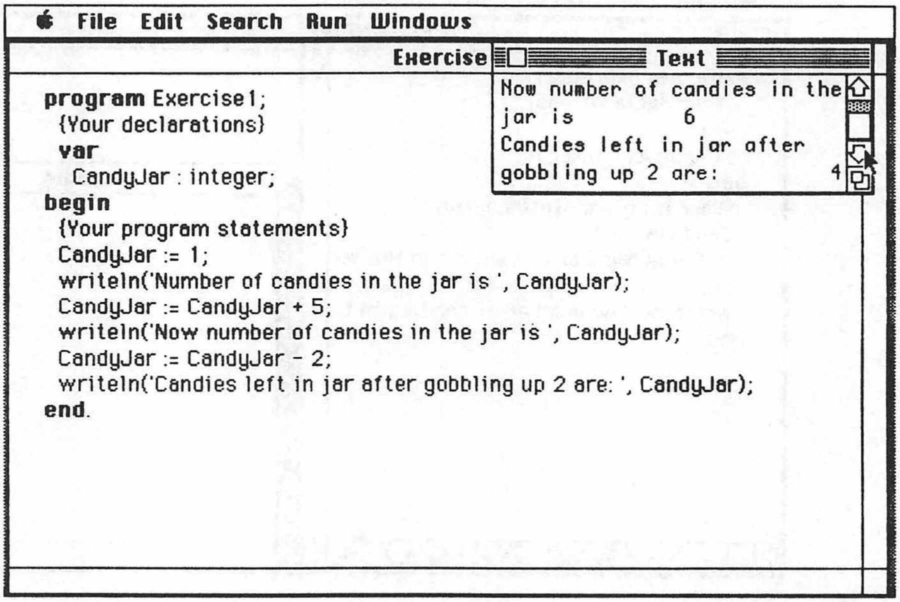
If you have not done so already, close program ProgTest by selecting CLOSE from the FILE menu. Select the DISCARD button if asked whether you want to SAVE or DISCARD ProgTest, since you will not need it later.

Once again, pull down the FILE menu, but this time select OPEN. Since Exercise1 is not on your system disk (we hope) it won't be listed in the dialog window among the files to OPEN. Therefore, click the EJECT button and insert the disk containing Exercise1. Now select Exercise1 by pointing and double-clicking it. Macintosh will ask you to re-insert the Macintosh Pascal diskette and then bring up Exercise1 in its Program Window. (See Figure 3.6.) Enlarge the Program Window, so you can see the entire program.

Recall that in Exercise1 you declared variable CandyJar, assigned one candy to it, and later added five more candies. What if you wanted to take some candy from it? (You certainly deserve a reward by now.) The following statement will do the job.

```
CandyJar := CandyJar - 2;
```

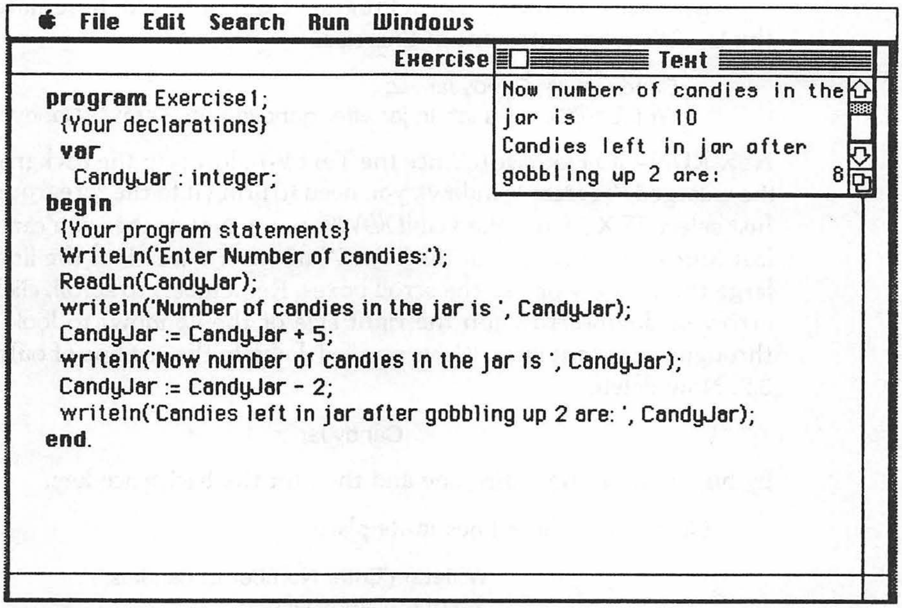
This assignment statement says to subtract 2 from CandyJar and copy the result back into variable CandyJar. Remember, the result of the computation performed to the right of the " := " is assigned to the variable on the left.



```
program Exercise1;
 {Your declarations}
 var
  CandyJar : integer;
begin
 {Your program statements}
 CandyJar := 1;
 writeln('Number of candies in the jar is ', CandyJar);
 CandyJar := CandyJar + 5;
 writeln('Now number of candies in the jar is ', CandyJar);
 CandyJar := CandyJar - 2;
 writeln('Candies left in jar after gobbling up 2 are: ', CandyJar);
end.
```

```
Now number of candies in the
jar is        6
Candies left in jar after
gobbling up 2 are:         4
```

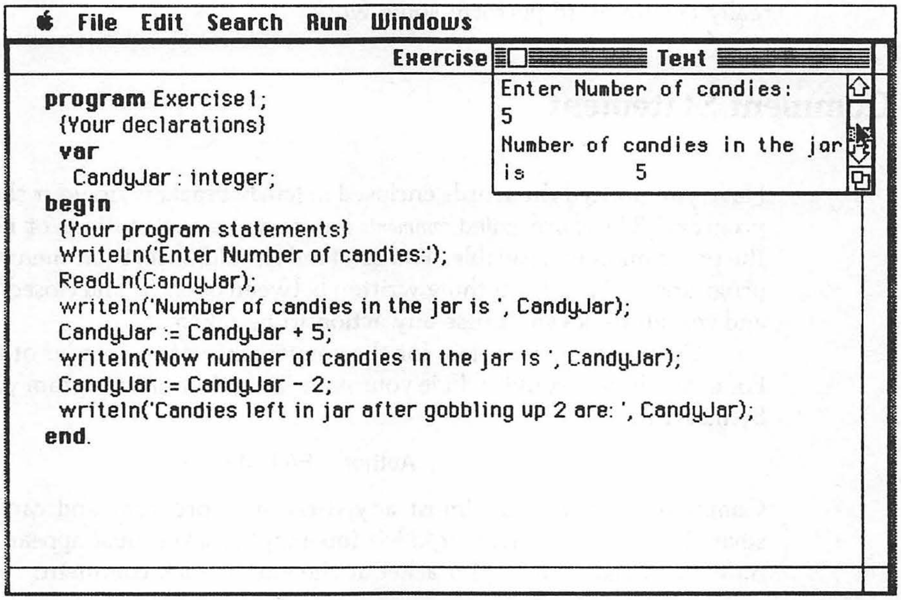
FIGURE 3.7
Exercise1, subtracting two gobbled candies.



```
program Exercise1;
  {Your declarations}
  var
    CandyJar : integer;
begin
  {Your program statements}
  WriteLn('Enter Number of candies:');
  ReadLn(CandyJar);
  writeln('Number of candies in the jar is ', CandyJar);
  CandyJar := CandyJar + 5;
  writeln('Now number of candies in the jar is ', CandyJar);
  CandyJar := CandyJar - 2;
  writeln('Candies left in jar after gobbling up 2 are: ', CandyJar);
end.
```

```
Now number of candies in the
jar is       10
Candies left in jar after
gobbling up 2 are:          8
```

FIGURE 3.8
Exercise1, use of ReadLn.



```
program Exercise1;
  {Your declarations}
  var
    CandyJar : integer;
begin
  {Your program statements}
  WriteLn('Enter Number of candies:');
  ReadLn(CandyJar);
  writeln('Number of candies in the jar is ', CandyJar);
  CandyJar := CandyJar + 5;
  writeln('Now number of candies in the jar is ', CandyJar);
  CandyJar := CandyJar - 2;
  writeln('Candies left in jar after gobbling up 2 are: ', CandyJar);
end.
```

```
Enter Number of candies:
5
Number of candies in the jar
is        5
```

FIGURE 3.9
Exercise1, first portion of Text Window.

Let's see how this works in Program Exercise1. Add these statements after the last WriteLn statement in Exercise1.

```
CandyJar := CandyJar -2;
WriteLn ('Candies left in jar after gobbling up 2 are: ', CandyJar);
```

Now RUN-GO Exercise1. Since the Text Window is in the background (behind the enlarged Program Window), you need to bring it to the foreground to read it. Just select TEXT from the WINDOWS menu. Notice that you can only see the last four lines of output in the Text Window. To see all of the lines either enlarge the window or use the scroll boxes. Remember, to scroll, click on the up-arrow or down-arrow (on the right side of the window) to look up or down through the output lines. We've scrolled down to the last line of output in Figure 3.7. Now delete:

```
CandyJar := 1 ;
```

by highlighting the entire line and then hit the backspace key.

Next insert these lines in its place:

```
WriteLn ('Enter Number of candies:') ;
ReadLn (CandyJar) ;
```

Make sure the Text Window is open, then run your program and enter a number when you are prompted by the blinking insertion point. (Don't forget to hit the return key after entering a number, or the Macintosh will never receive your message!) Compared to previous versions of Exercise1, the current one is really beginning to perform some work.

Comment Statement

Have you noticed the words enclosed in {curly brackets} in your skeleton Pascal program? These are called *comments* and serve as explanations or notes to make the program more readable. You can (and should) insert comments within your programs. In Pascal, anything written between open "{" and closed "}" is ignored and therefore doesn't cause any action to be taken.

Comments exist solely for the convenience of the reader of the program. For example, you could include your name as author of a program you've written by inserting:

```
{ Author : FAT MAC }
```

Comments can appear almost anywhere in a program and can extend over several lines. Just remember, a left (opening) bracket must appear at the beginning and a right (closing) bracket at the end of each comment:

```
{Program Sweet Tooth
        by
     FAT MAC}
```

Summary

In this session you learned about the structure of a simple Pascal program (see Figure 3.10). *Constants* and *variables* must be defined in the *declaration part* of the program. Constants are defined using their exact value (explicitly), which cannot be altered by any subsequent action in the program.

Variables are assigned their values using *assignment* or *ReadLn* statements. Variable values can "vary" or be changed by the executing program. The value of a variable is processed by performing mathematical computations using assignment statements.

Begin and *End* enclose all statements in the program body; semicolons separate all program statements; and a period always follows the end of the program.

The output of a running program appears in the Text Window through either the Write or WriteLn statement. WriteLn sends a Return to the Text Window and Write does not.

We have not studied examples of all operators you may use in an expression, but for quick reference later on, see Table 3.1, Precedence Table. The operators at the top of the list are done first, followed by the second precedence operators and so on.

```
program identifier;
 const                        }
constant definition;          }

                              }
  constant definition;        }  declaration part
 var                          }
  variable declaration;       }
  variable declaration;       }
 begin                        }
  program statement;          }  program body
  program statement;          }
 end.                         }
```

FIGURE 3.10
A complete program structure.

Table 3.1 PRECEDENCE TABLE

Operator	*What It Does*	*When It Is Done*
@, —, not	address, negative, Boolean NOT	FIRST
*, /, div, mod, and	multiply, divide, divide modulo, Boolean AND	SECOND
+, —, or	add, subtract, Boolean OR	THIRD
=, <>, <, >,	comparisons	LAST

Problem Solving

1. How do you define an object as a variable to represent the value PI?
2. What is an "identifier"?
3. What is a "reserved word"?
4. What is the purpose of semicolons?
5. What is a "string constant"?
6. Explain why the contents of a variable can be changed.
7. Write a "WriteLn" statement to print "IT'S TIME TO PARTY."
8. What is the difference between constants and variables?
9. Write a program to add two numbers, B and C, together. B and C get their values from the keyboard, and are integers. Display the answer in the Text Window.
10. Modify program Exercise1 to subtract Grab from CandyJar. Grab is entered from the keyboard. Display the new value of CandyJar in the Text.

Session 4:

Integer, Real, and Char Expressions

In this session you will learn how to write integer, real, and char expressions that evaluate to integer, real, or char values. Expressions can be used as part of assignment, Write, WriteLn, and other statements covered in later sessions.

The Instant Window

A quick way to see how Pascal interprets a certain statement or expression is to open the Instant Window and use it to evaluate one or more statements. The *Instant Window* can be used to enter statements that are not part of a program residing in the Program Window. Or you can halt a running program, open the Instant Window, execute a few statements, and then resume execution of the program. Alternately, the Instant Window can be used to run tests, as we will do later in this session.

Start Macintosh Pascal and close the Program and Drawing Windows. This leaves the Text Window and a blank screen. Select the WINDOWS- INSTANT menu and get the Instant Window as shown in Figure 4.1. We will use the two windows shown in Figure 4.1 throughout the following hands-on exercise.

The DO IT button in the Instant Window is for executing whatever happens to be in the Instant Window. Initially, "{Any statement, any time}" appears as a selected comment. Press the backspace key to erase this comment.

Enter the following output statement into the Instant Window and click DO IT. Watch what appears in the Text Window.

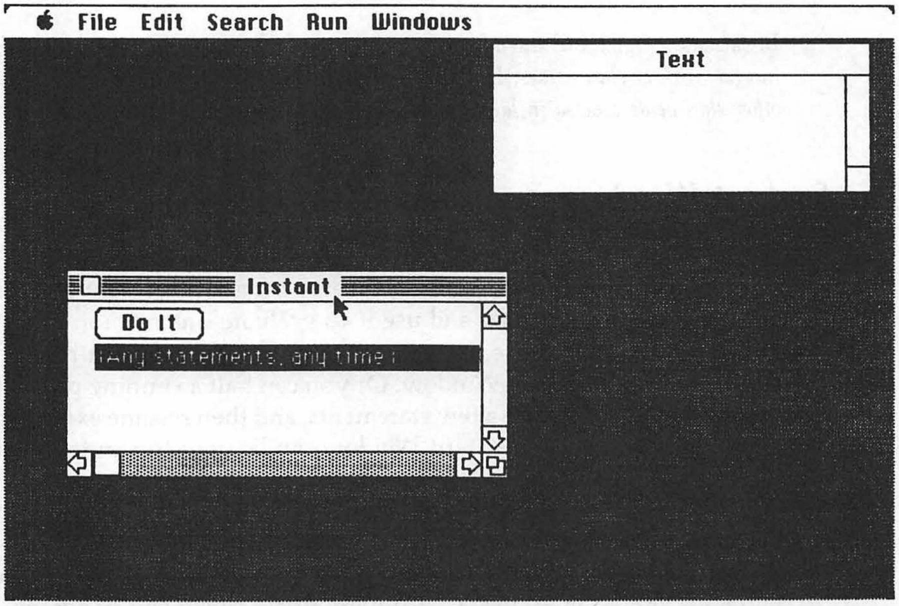
```
WriteLn ('Greetings from Instant')
```

As you can see, this single statement is executed and its output message appears in the Text Window just as if it were a running program. During this session you will be executing WriteLn statements containing expressions like the one below (see Figure 4.2).

```
WriteLn (10 * 3) ;
```

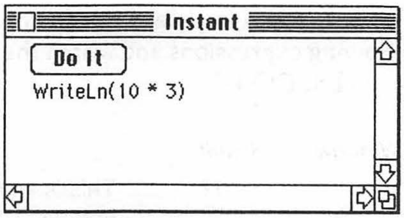
The value of this expression is 30. The WriteLn statement causes 30 to be output to the Text Window.

The idea of using an expression within a WriteLn statement will be used in examples throughout this session. To change an expression to some other test expression, select (highlight) the expression by dragging the mouse over it, and type in a replacement.



```
{Any statements, any time!
```

FIGURE 4.1
Instant Window and Text Window.



```
WriteLn(10 * 3)
```

FIGURE 4.2
Instant Window with a single statement.

Integers

Integers are the simplest kind of numbers stored in a Pascal program. They range in value from -32,768 to 32,767. You can add, subtract, multiply, divide, and perform I/O on integers using the following operators.

+, -, *	add, subtract, multiply.
div, mod	divide (quotient), divide (remainder).

In order to use these operators in expressions, you must follow the rules of hierarchy or ordering as you evaluate different operations. There are two levels of hierarchy and precedence of operations.

+ / **div mod**	these are done first . . .
+ -	followed by these,

When more than one operator from one level of hierarchy is found in an expression, they are evaluated from left to right. Using parentheses can change this order of evaluation, as well as make it clearer.

In addition to these operators, the following *intrinsic functions* apply to integers.

ReadLn ( )	Read an integer from the keyboard.
WriteLn ( )	Write to the Text Window.
abs ( )	Absolute value of.
sqr ( )	Square of.
sqrt ( )	Square root of.
chr ( )	Character of.
succ ( )	Successor of.
pred ( )	Predecessor of.

Use the mouse and keyboard to edit the WriteLn statement of Figure 4.2. Enter the following expressions and watch the result appear in the Text Window each time you click DO IT.

Instant Window	*Result*	*Explanation*
10 + 2	12	This is an easy one. The sum of 10 and 2 is 12, but notice how many leading blanks appear in front of 12 in the Text Window.
10 + 2 :3	12	This time only 3 columns are allowed for 12. The :3 specifies the number of columns allowed for a result when displayed from WriteLn.
5 div 2	2	Div means "quotient of". The quotient of 5 divided by 2 is 2.
5 mod 2	1	The remainder after the division is 1. Mod means to take the remainder (of 5 divided by 2).
-5 div 2	-2	The quotient of (-5) divided by 2.
-5 mod 2	1	The remainder after (-5) is divided by 2. (It will always be positive because (-2) * (2) = (-4).
5 mod (-2)	Error	You get a bug because this is undefined.
- (5 mod 2)	-1	The result of 5 mod 2 is made negative.
11 * 3	33	Multiplication is as usual.
11 * 3 div 2 + 1	17	11 * 3 is done first, then div 2, and lastly + 1.
11 * (3 div 2) + 1	12	3 div 2 is done first because of ( ).
abs (-3)	3	The absolute value ignores sign.
sqr (-3)	9	This is the same as (-3) * (-3).
chr (65)	A	The integer is converted into a character. The ASCII code for 'A' is 65.
succ (1)	2	The successor to 1 is 2. The next larger value is one greater than 1.

pred (2)	1	The predecessor of 2 is 1. The next smaller value is one less than 2.

Try each of these expressions on your computer. After entering a new expression, click DO IT and watch the Text Window. Click the Instant Window once to make it active following DO IT. Drag the mouse over the previous expression; then enter the next expression.

Integer expressions are typically found in assignment statements. Suppose X is an integer.

```
X := 10 div 3;
```

Integer variables can be used freely in place of integers in either assignment or WriteLn statements. Keep in mind that these variables *must* have been declared previously in the Var statement.

```
X := Y div Z;
```

The quotient of Y divided by Z is stored in X. This causes the previous value in X to be lost.

Reals

Now erase the integer expression in the WriteLn statement of Figure 4.2 and try the following operations on real numbers. You may be surprised by the "e" format obtained when real values are written to the Text Window.

```
1.5e + 2
```

This format is actually a method for expressing either very small or very large numbers and is called *scientific notation*. The "e" stands for "times 10 to a power of" whatever number follows. This means the decimal point is to be moved to the right (+) or left (–) a certain number of digits. Thus 150.0 is equivalent to 1.5e + 2 and 0.15 is equivalent to 1.5e –1.

The "e" separates the number from the exponent part which tells how many digits to move the decimal point. Remember, whenever a real number is written in a Pascal statement, if it contains a decimal point there must always be a leading digit to the left of the decimal (even if it is zero). Therefore, .5 is an error and 0.5 is legal.

Instant Window	*Result*	*Explanation*
123.45	.12e + 2	The output is displayed in "e format." Only two significant places are shown.
123.45 :7:2	123.45	The output is formatted as follows: 7 columns for the entire number, 2 columns for the decimal fraction.

10 / 3 :7:2	3.33	The result of dividing two integers is a real number. This is called *type coercion* because the integers are changed into reals, the division is done, and a real result is obtained. The format :7:2 forces the WriteLn to display two decimal fraction digits.
1.0 / 2.0 :7:2	0.50	The result of dividing 1.0 by 2.0 is 0.50. Without the :7:2 format, the result would be displayed in e format.
abs (–5.5) :7:2	5.50	Absolute value discards the minus sign.
sqr (–5.0) :7:2	25.00	Square the number.
sqrt (30) :7:2	5.48	The square root of the number. Notice the 30 is coerced.
trunc (5.6)	5	Converts 5.6 to an integer by truncating its decimal fraction part.
round (5.6)	6	Converts 5.6 to an integer by rounding it off.
sin, cos, arctan		. . . three trigonometric operators. The real number is in units of radians, i.e., sin (3.14159) is 2.7e -6, close to zero.
exp (2)	7.4e + 0	The constant e = 2.7 . . . raised to the power of 2.
ln (7.4e + 0)	2.0e + 0	The inverse of exp. The natural logarithm of . . .
3.1 + 4 / 2 * 6	1.5e + 1	Evaluation is from left to right unless a higher priority operation is to be done. In this example, the 4 / 2 is done first.
3.1 + (4 / 2) * 6	1.5e + 1	Same as above except the meaning is made clear by the ( ). (4 / 2) is done first, * 6 is second, and 3.1 + is last.
3.1 + 4 / (2 * 6)	3.4e + 0	The parenthetic expression is done first. Then 4 / 12 is done, followed by 3.1 +. The division coerced 4 and 12.
(3.14 + 4) / 2 * 6	2.1e + 1	The parenthetic expression is done first. Then 7.1 is divided by 2. Finally, * 6 is performed.

(3.1 + 4) / (2 * 6)	5.9e -1	The meaning is clear! The numerator is computed, and the result is obtained by division. The (3.1 + 4) sum is obtained by coercing 4 into 4.0.
(3.1 + 4) / (2 * 6) :8:3	0.592	The answer is formatted.

The important thing to remember about real expressions is that they cause integers to be coerced into reals whenever necessary. This is especially important to recall when mixing reals and integers in assignment statements.

If X : Real and Y : Integer have been declared in your program, then

```
X := Y;
```

is legal, but

```
Y := X;
```

is an error! This is because an integer can be coerced into a real, but a real cannot be coerced into an integer. To get around this problem, you must use trunc or round.

```
Y := trunc (X) ;
Y := round (X) ;
```

The second lesson to be learned from the exercises above is the importance of parentheses. Always use parentheses to make sure the computer does the calculations exactly the way you intend them to be done. A minor slip-up can mean hours of debugging time.

Char

The character set shown on your keyboard is encoded (internally) by associating a number with each character. Thus characters can be manipulated much like integers. The ASCII code below is the character set used by Macintosh; it contains 95 printable characters and many special (unprintable control) characters. Notice the first character is a space.

ASCII Character Set

```
  ! " # $ % & ' ( ) * + - , . / 0 1 2 3 4 5 6 7 8 9 : ; < = > ? @
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z [\] ^ _ '
    a b c d e f g h i j k l m n o  p q r s t u v w x y z {  }
```

This is made possible by converting the char value or variable into an integer. The following examples should be run using the Instant Window.

Instant Window	*Result*	*Explanation*
ord ('A')	65	The internal numeric code for capital A.
ord ('a')	97	The internal numeric code for lower case a.
ord (' ')	32	Even a blank space has an internal numeric code.
succ ('a')	b	the successor of a is b. The alphabet is ordered.
pred ('b')	a	The predecessor of b is a.
succ ('ab')	Error	A char is a single character.
ord ('Z') - ord ('A')	25	There are 25 letters of the alphabet between A and Z.
chr (ord ('H'))	H	chr and ord are inverses of one another.

Char variables must be declared in the Var statement. Any char value can be assigned to a char variable. If Ch : Char and X : Integer are declared, then the following is a valid assignment statement.

```
Ch := Chr (X) ;
```

The value of Ch may be displayed from within a WriteLn statement.

```
WriteLn ('Ch) ;
```

Integers and chars are called *ordinal types* because they can be matched with whole numbers and counted. Real is not an ordinal type because there are infinitely many real numbers, too many to be counted and matched. The following statement is invalid because X : Real cannot be matched up with an integer.

```
WriteLn ( Ord (X) ) ;
```

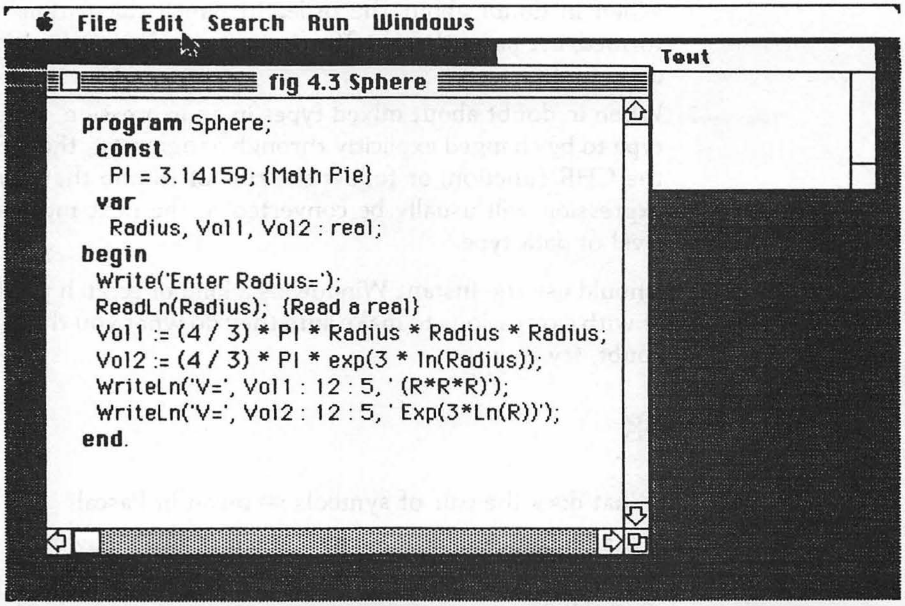
The difference between a real and an ordinal causes many inconsistencies in Pascal.

In general, be wary of mixing reals and ordinals in expressions (coercion), using reals in the same or similar places along with ordinals. Also, avoid comparing or converting reals to ordinals and the reverse.

A Hands-On Example

This session is concluded with a hands-on example you can run on your Macintosh. Enter the program shown in Figure 4.3 and RUN-GO. Compare the two values obtained from the two expressions for the volume of a sphere.

$$\text{Volume of sphere} = \frac{4 \text{ pi } R^3}{3}$$



```
program Sphere;
 const
  PI = 3.14159; {Math Pie}
 var
  Radius, Vol1, Vol2 : real;
begin
 Write('Enter Radius=');
 ReadLn(Radius);  {Read a real}
 Vol1 := (4 / 3) * PI * Radius * Radius * Radius;
 Vol2 := (4 / 3) * PI * exp(3 * ln(Radius));
 WriteLn('V=', Vol1 : 12 : 5, ' (R*R*R)');
 WriteLn('V=', Vol2 : 12 : 5, ' Exp(3*Ln(R))');
end.
```

FIGURE 4.3
Program to compute the volume of a sphere.

The expression for Vol1 computes R-cubed by repeated multiplication, but the same result is obtained using exp and ln. Vol2 should equal Vol1 because

Exp (3 * Ln (Radius))

is exactly the same as

Radius * Radius * Radius

Notice that Pascal does not have an exponentiation operator; therefore, to raise a value or a variable to a power, you must use the form shown in Vol2. In general, to raise R to the power P, use the following expression.

Exp (P * Ln (R))

This works even when P is not an integer. Hence, if P is less than one, a root is obtained, and if P is greater than one, a power is obtained.

Summary

In this session you learned how to form elementary expressions. These rules will be used frequently throughout the remainder of this book. There are two fundamental things you should remember:

1. When in doubt about the order in which calculations will be performed, use parentheses to force the parenthetic subexpressions to be done first.
2. When in doubt about mixed types in an expression, simply force the type to be changed explicitly through assignment, the ORD function, the CHR function, or follow the rule of thumb that says the mixed expression will usually be converted to the next more sophisticated level of data type.

You should use the Instant Window as a kind of scratch pad that lets you experiment with expressions to make sure they do what you think they should. When in doubt, try it out!

Problem Solving

1. What does the pair of symbols := mean in Pascal?
2. What is the value computed by each of the following?
   a. 4 / 2
   b. 4 div 2
   c. 4 mod 2
   d. Exp (3 * Ln (2))
   e. 3.12 / 1.6
   f. 2 + 3 / 3 + 2
   g. 2 * 3 / 3 * 2
   h. 8 mod 5 div 2
3. Enter the following expressions into a WriteLn statement and find out what they write to the Text Window.
   a. 123 :2
   b. -85.6 :5:0
   c. round (1.56e + 1) :12:5
   d. 13 mod 4 div 3 :10
   e. Exp (5 * Ln(2)) :10
4. Write programs to compute the following areas:
   a. circle with radius R
   b. cylinder with radius R and height H
   c. cube with side S
   d. right triangle with base B and height H
5. Use the Instant Window to see what the following expressions evaluate to:
   a. 1 + 1 / 1
   b. Ord ('$')
   c. Chr (Ord ('A') + 32)
   d. Ord ('1') :10
   e. Ord ('1') :1
   f. What is the difference between (d) and (e) ?

Session 5:

What Is a Subprogram?

In this session you will learn how to use small subprograms to refine larger programs. The idea is very simple: Large programs are composed of small programs in a kind of "divide and conquer" approach to programming. Pascal has two kinds of subprograms: procedures and functions.

Pascal Subprograms

Now that you have written a small program you will be eager to write larger and more complex programs. As your programs grow in size and complexity, they will become difficult to understand and write correctly. Small modifications will "ripple" through the entire program and cause bugs in the least expected places.

To combat rising complexity and to reduce the likelihood of errors, professional programmers have developed techniques for keeping programs as small as possible. One of the most successful techniques is called *program refinement,* and one of the most common units of program refinement is the procedure.

A *procedure* is a block of text containing local data declarations and a body of instructions. It is very much like a miniature program.

A *function* is a block of text containing local data declarations and local instructions like procedures, but unlike a procedure, a function takes on a value much like a variable. The difference between a function and a procedure lies in how they are used and how they return computed values to the main program.

A procedure represents an *action,* and its name in a program tells the computer to perform one or more operations on data passed to the procedure. A function, on the other hand, must *represent a value,* and its name in a program tells the computer to calculate and store a value. Both procedures and functions are used in a style of programming called *modular programming.*

Procedures

Suppose you want to compute the miles per gallon achieved on a vacation trip. The formula for miles per gallon is

MPG = MILES / GALLONS

Given the number of miles traveled and the number of gallons consumed, the value of MPG is calculated by simple division.

A program to compute MPG must do some other things as well. Here is a "to do" list of the steps in computing MPG.

1. Get the numerical values of MILES and GALLONS from the user.
2. Perform the division shown in the formula.
3. Display the results, including the values of MILES and GALLONS.

A "to do" list should give you a good idea of what steps must be done in the body of a program, but it does not reveal anything about the data to be used. Recall that a Pascal program consists of both a data declaration section and a program body. Therefore, in addition to the three steps above, you must include a data declaration section.

A. MILES is a real number, initially input.

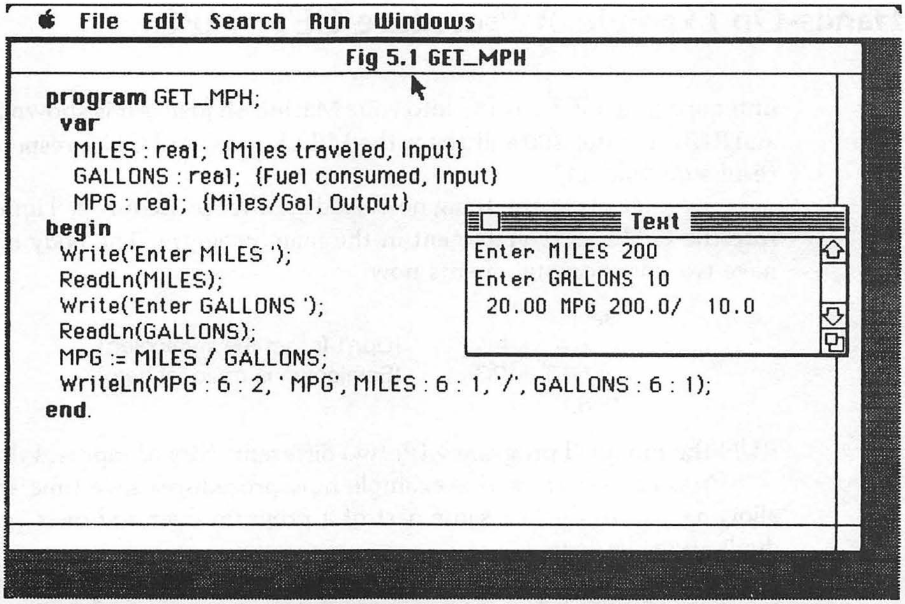
B. GALLONS is a real number, initially input.

B. MPG is a real number to be computed and then output.

The data declaration specification lists what each variable is and what it does. This input/output status of each object is important because the Pascal program must specify what the user is to enter and what the computer is to display. The complete program in Figure 5.1 computes what we want.

In program GET_MPG the output is formatted using :6:2 and :6:1 following the variables in the final WriteLn statement. Recall that the first digit means to use six columns for the entire number including the sign, decimal point, and decimal fraction. The second number indicates how many digits are to be displayed to the right of the decimal point. Therefore, :6:2 means to display a number such as 18.57 or -20.00 (two decimal places) and :6:1 specifies numbers such as 256.5 and -999.9 (one decimal place).

A procedure is like a miniature program since it has local data and local instructions in its body. To convert GET_MPG from a program into a procedure, do two things:



```
program GET_MPH;
 var
   MILES : real; {Miles traveled, Input}
   GALLONS : real; {Fuel consumed, Input}
   MPG : real; {Miles/Gal, Output}
begin
  Write('Enter MILES ');
  ReadLn(MILES);
  Write('Enter GALLONS ');
  ReadLn(GALLONS);
  MPG := MILES / GALLONS;
  WriteLn(MPG : 6 : 2, ' MPG', MILES : 6 : 1, '/', GALLONS : 6 : 1);
end.
```

```
Enter MILES 200
Enter GALLONS 10
 20.00 MPG 200.0/  10.0
```

FIGURE 5.1
Program GET_MPH.

1. Change "Program" to "Procedure" in the program header, and change the ending period into a semicolon.
2. Insert "Procedure GET_MPG" inside another program (immediately before the Begin), and then "call" it from somewhere inside the other program.

Figure 5.2 shows how GET_MPG is used from within another program called MAIN. First, the entire block of text called Procedure GET_MPG is inserted into the data declaration section of Program MAIN. Second, the actions of GET_MPG are invoked by "calling" GET_MPG from within the body of MAIN.

Procedure Call: A procedure is *called* by using its name as if it were a valid statement in Pascal.

Notice the similarity between a procedure call and a Write, WriteLn, and ReadLn statement. Actually, Write, WriteLn, and ReadLn are special procedures called *intrinsic* procedures because they are built into Pascal. An intrinsic procedure is one that is already defined in Pascal. Intrinsic procedures are called like any other procedure, but since they are already defined by the language, you do not need to insert them in the data declaration section of your main program.

Hands-On Example of Procedure GET_MPG

Enter program GET__MPG into your Macintosh just as it is shown in Figure 5.2 and RUN it. Enter 200 and 10 for the MILES and GALLONS, respectively. What results do you get?

Now let's try something new. Add another procedure call immediately before the GET_MPG statement in the main program. The body of MAIN will have two identical statements now.

```
Begin
   GET_MPG ;        {Don't forget the semicolon}
   GET_MPG          {Semicolon is optional here}
End.
```

RUN the modified program with two different pairs of inputs. What happens?

You can see from this example how procedures save time and effort by allowing you to call the same part of a program over and over again without duplicating the text.

Procedures are important for a variety of reasons, but perhaps the most important feature of a procedure is its ability to compartmentalize a section of your program. Procedures act as fences or boxes which contain their own data and instructions for doing a specific operation. Procedures should be used to do a well-defined operation on well-defined input and output data.

The GET_MPG example can be improved by clearly defining all of its inputs and outputs. Here is how to improve GET_MPG.

1. Remove the Var statements from GET_MPG by marking them with the mouse (drag the mouse across Var, MILES:Real; etc.) Press the backspace key to delete them.
2. Change "Procedure GET_MPG;" to the following:

   ```
   Procedure GET_MPG (MILES,GALLONS:Real ; Var MPG:Real) ;
   ```

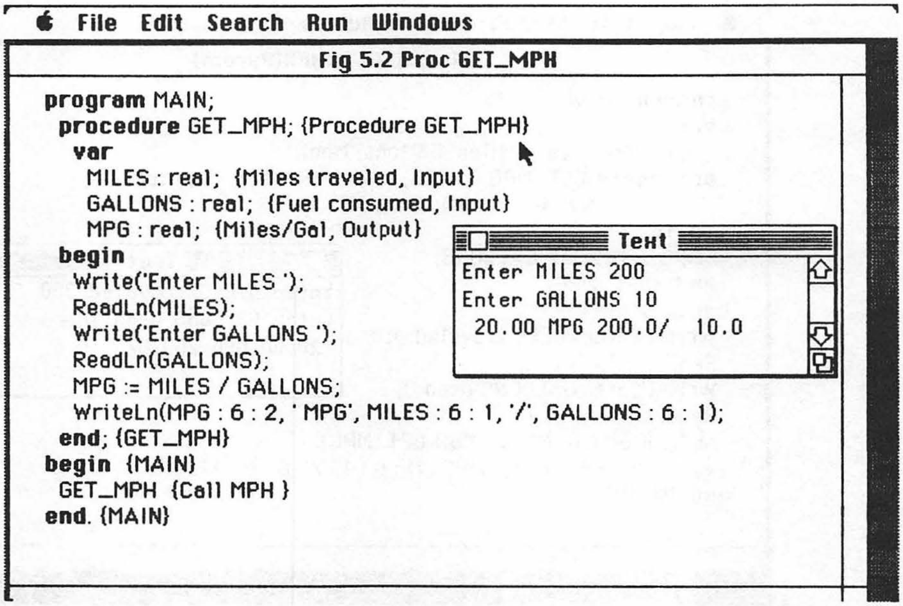
   In other words, add the parenthetic list of objects to the procedure heading.
3. Change the call statement in the body of MAIN to the following:

   ```
   GET_MPG (M,G,MPG)
   ```

   Add the parenthetic list of objects to the call statement.
4. Add the following declaration to the main program, immediately after the program statement:

   ```
   Var
      M,G,MPG : Real ;          {commas are a shortcut}
   ```

   Notice that commas have been used to separate names in a list. This is a shortcut way to declare groups of variables which are all of the same type.



```
program MAIN;
 procedure GET_MPH; {Procedure GET_MPH}
  var
   MILES : real; {Miles traveled, Input}
   GALLONS : real; {Fuel consumed, Input}
   MPG : real; {Miles/Gal, Output}
 begin
  Write('Enter MILES ');
  ReadLn(MILES);
  Write('Enter GALLONS ');
  ReadLn(GALLONS);
  MPG := MILES / GALLONS;
  WriteLn(MPG : 6 : 2, ' MPG', MILES : 6 : 1, '/', GALLONS : 6 : 1);
 end; {GET_MPH}
begin {MAIN}
 GET_MPH {Call MPH }
end. {MAIN}
```

```
Enter MILES 200
Enter GALLONS 10
 20.00 MPG 200.0/  10.0
```

FIGURE 5.2
Procedure GET_MPG.

5. Finally, erase all of the Write, WriteLn, and ReadLn statements from the body of GET_MPG and insert the following, immediately before the call statement in the main program. (The final program is shown in Figure 5.3.)

```
Write ('Enter miles traveled') ;
ReadLn ( M ) ;
Write('Enter gallons used') ;
ReadLn ( G ) ;
```

Then insert the following immediately after the GET_MPG call:

```
WriteLn (MPG:6:2,' MPG', M:6:1,'/',G:6:1) ;
```

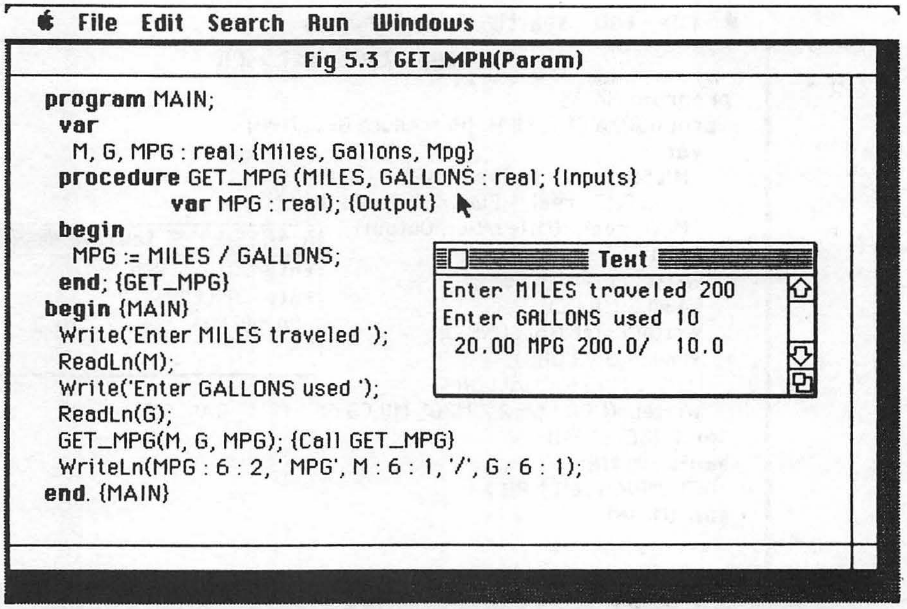
These steps have made GET_MPG a well-defined block of Pascal code for computing mileage from two well-defined input values. The inputs and outputs are called *procedure parameters,* and they are enclosed with parentheses.

The procedure heading contains a list of input and output parameters. The first two are inputs:

```
MILES,GALLONS:Real ;
```

The third parameter is an output parameter, and has a reserved word Var in front of it to designate it as such.

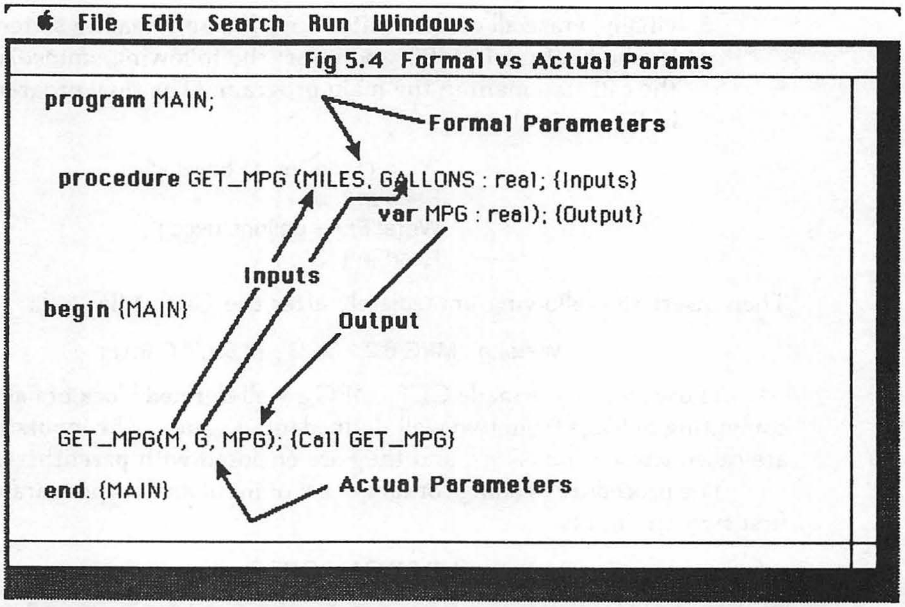
```
Var MPG:Real ;
```



```
program MAIN;
 var
  M, G, MPG : real; {Miles, Gallons, Mpg}
 procedure GET_MPG (MILES, GALLONS : real; {Inputs}
         var MPG : real); {Output}
 begin
  MPG := MILES / GALLONS;
 end; {GET_MPG}
begin {MAIN}
 Write('Enter MILES traveled ');
 ReadLn(M);
 Write('Enter GALLONS used ');
 ReadLn(G);
 GET_MPG(M, G, MPG); {Call GET_MPG}
 WriteLn(MPG : 6 : 2, ' MPG', M : 6 : 1, '/', G : 6 : 1);
end. {MAIN}
```

```
Enter MILES traveled 200
Enter GALLONS used 10
 20.00 MPG 200.0/  10.0
```

FIGURE 5.3
Program to compute miles per gallon.



```
program MAIN;

 procedure GET_MPG (MILES, GALLONS : real; {Inputs}
           var MPG : real); {Output}

begin {MAIN}

 GET_MPG(M, G, MPG); {Call GET_MPG}
end. {MAIN}
```

FIGURE 5.4
Actual vs. formal parameters and input vs. output variables.

Therefore, in a procedure heading, all input and output variables must be declared inside parentheses, and all output variables must have a leading Var Reserved Word.

Procedure Parameters: A list of input and output variables appearing in the procedure heading. The output variable must be preceded by the reserved word Var. The list of variables that appear in the procedure heading statement are called the *formal parameters,* and the list of variables appearing in a procedure call statement are called *actual parameters.* The diagram in Figure 5.4 illustrates the relationship between formal and actual parameters.

Notice in Figure 5.4 that it is possible for a formal and actual parameter to have the same name. MPG is the name of both the formal and actual parameters (output variable). This is like two people having the same last name. The full name of these two variables shows that they are different: MAIN MPG is the full name of the actual parameter; GET_MPG is the full name of the formal parameter. They are as different as Ted and Molly Lewis.

Even though the names may differ, the values passed back and forth are the same. GET_MPG is calculated inside GET_MPG and its value is passed to MAIN MPG. The value is either *copied* or *shared,* depending on whether the formal parameter is prefixed with Var (shared) or not (copied). For more information on how procedures and functions communicate, see Session 14.

Run the modified program shown in Figure 5.3. How would you change it to get two mileage calculations?

Procedures behave like statements in a Pascal program. They can process multiple inputs and return multiple outputs back to the calling main program. There is another form of subprogram called a *function* which behaves like a variable. A function is a special kind of subprogram that returns a single value corresponding with its name.

Functions

Functions are subprograms consisting of a data declaration section and a body just like a procedure. Functions can have zero or more input parameters, but only one output variable. Furthermore, the output variable of a function is the name of the function itself.

The function name is used just like a name of a variable. This means that a function name may appear in an expression, for example, as part of an assignment statement or inside a WriteLn statement.

Function Call:	A function subprogram is called by using its name in an expression, parameter list, or most places where a variable may be used.

There are several notable exceptions to the rule above: Function names may not be used in a list of variables in a ReadLn statement; the function name may not appear on the left side of an assignment statement, except from within the function itself; and so on. We will point out these exceptions as the need arises.

Since functions behave like variables they must be declared as certain types. The heading statement of a function differs from the heading of a procedure in two respects. First, the word "Function" appears instead of "Procedure," and second, the function name must be given a type.

```
Function Mileage (MILES, GALLONS:Real) : Real ;
```

The two input values, MILES and GALLONS, are used to compute MILEAGE. Notice how MILEAGE is made into a *real* valued variable. MILEAGE is the output value returned to the main program.

Hands-On Example of Function MILEAGE

Suppose the previous example is used to show the difference between a function and a procedure. The GET_MPG procedure can be changed into a function called MILEAGE by following the steps below (use the program in Figure 5.3 as your starting point).

1. Change the following statements of Procedure GET_MPG.

   a. Remove the procedure heading and put in its place:

   ```
   Function MILEAGE ( MILES, GALLONS:Real ) :Real ;
   ```

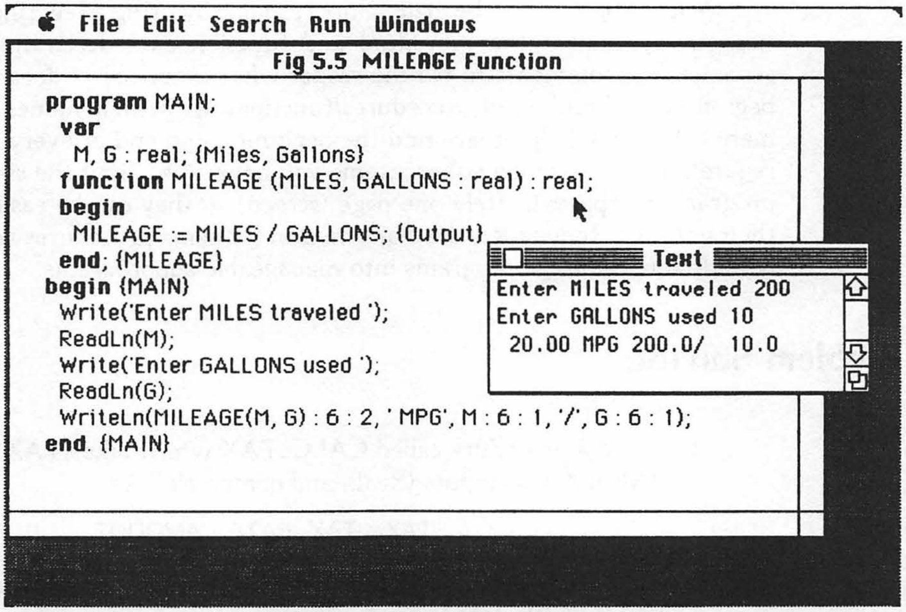
   b. Change the assignment statement to:

   ```
   MILEAGE := MILES / GALLONS;
   ```

2. Change the main program by removing MPG from the list of variables and rewriting the WriteLn statement, as follows. (Also, remove the GET_MPG (M,G,MPG); statement because it is no longer needed.)

   ```
   WriteLn(MILEAGE(M,G) :6:2, ' MPG',M:6:1, '/',G:6:1) ;
   ```

   Notice how MILEAGE is used: the function is called from within the WriteLn statement. The value of MILEAGE is computed in Function MILEAGE and then returned to the WriteLn statement. The WriteLn statement outputs this value according to the format :6:2.



```
program MAIN;
  var
    M, G : real; {Miles, Gallons}
  function MILEAGE (MILES, GALLONS : real) : real;
  begin
    MILEAGE := MILES / GALLONS; {Output}
  end; {MILEAGE}
begin {MAIN}
  Write('Enter MILES traveled ');
  ReadLn(M);
  Write('Enter GALLONS used ');
  ReadLn(G);
  WriteLn(MILEAGE(M, G) : 6 : 2, ' MPG', M : 6 : 1, '/', G : 6 : 1);
end. {MAIN}
```

```
Enter MILES traveled 200
Enter GALLONS used 10
 20.00 MPG 200.0/  10.0
```

FIGURE 5.5
Mileage function.

Figure 5.5 shows the complete program including Function MILEAGE as it should appear before you attempt to RUN it. Try it on the following input pair: 350 and 15.

Functions return a single value, accept zero or more input parameters, and work very much (but not exactly) like variables. Functions compartmentalize frequently used calculations and should be used where a single value is computed and then returned to the main program.

Summary

You can have as many functions and procedures as needed in a main program. They are declared after the Var list and before the Begin reserved word in the main program.

It is customary to use verb phrases for names of procedures because procedures perform actions similar to Pascal statements. Hence, GET_MPG, WRITE_CHECK, CALCULATE_TAX, and other "action phrases" should be used consistently. Similarly, it is best to use noun phrases for variable names and function names. MILEAGE, GALLONS, MPG, and phrases such as PROPERTY_TAX or HOUSE_PAYMENT are preferred for variables and functions.

Your programs can be made more readable by using a few rules of aesthetics. Always indent the procedure and function data declaration and body sections. Use comments to tell the reader what every object does. Bracket the beginning and ending of procedures/functions with their names inside comments. This will help others find the beginning and end of every subprogram regardless of how many subprograms you have. Also, limit the size of all subprograms to approximately one page (screen) so they can be easily viewed in their entirety. Remember, the major reason for using procedures and functions is to decompose large programs into manageable subprograms.

Problem Solving

1. Write a procedure called CALC_TAX which takes TAX_RATE and AMOUNT as inputs (Real), and computes TAX.

   TAX = TAX_RATE * AMOUNT

2. Modify the procedure of Problem 1 to make a function called TAXES (Real) which does the same thing.
3. George can mow his lawn in G = 3 hours by himself. Together, George and Mary can mow the lawn in GM = 2 hours. How long would it take for Mary to mow the lawn by herself? Write a Pascal program that accepts values of G and GM as inputs, and calculate M = time for Mary to mow the lawn by herself, as output. Use a function called MARYS_TIME to do the calculations. (Hint: When G = 5, GM = 2, then MARYS_TIME should be 3.333 hours.)
4. The area and perimeter of a rectangle are computed as follows:

   AREA = LENGTH * WIDTH
   PERIMETER = 2 * (LENGTH + WIDTH)

   Given integer values of LENGTH and WIDTH, write a procedure to compute AREA and PERIMETER.

   Why is it not possible to use a single function to solve this problem?
5. Write a procedure called GET_REAL which always writes a prompt, 'Enter Real number', and then gets a real number called R_NUMBER from the user. R_NUMBER is returned as an output parameter of GET_REAL.
6. Use the idea in Problem 5 to implement procedures GET_INTEGER and GET_REAL for entering Integers and Reals. Use GET_INTEGER to solve Problem 4 and GET_REAL to solve Problem 3.

Session 6:

Choosing in Pascal

In this session you will learn how to write programs which use Boolean values, variables, and expressions to control decision making from within a running program. Additionally, IF statements containing Boolean expressions and case statements will be illustrated in several hands-on examples. Finally, RUN-STEP, RUN-STOPS IN, RUN-STOPS OUT, and the Observe Window of Macintosh Pascal will be introduced.

Boolean Variables

George Boole (1815–1864) was a British mathematician who invented Boolean algebra—a peculiar form of algebra in which variables take on one of two possible values. A *Boolean constant* is either True or False; a *Boolean variable* is a variable whose type is Boolean.

One hundred years later, an American mathematician named Claude Shannon used Boolean algebra to show how an electronic digital computer might simulate human decision making. The idea was to model Yes-No decisions with Boolean variables and then build an electronic circuit to "remember" the Yes-No value of each Boolean variable. This later became the basis of all electronic binary computers.

In Pascal a variable is of type Boolean if it is declared Boolean.

```
Var
      P : Boolean;
```

The only value allowed for P is either True or False. The simplest way to establish the value of P is to assign True or False to it directly.

```
P := True;            { Assign TRUE to P }
```

True and False are constants just like 0, 5.3, and 'A'. True and False should not be used as names of objects in Pascal.

The constants True and False may be entered in a running program through the ReadLn procedure and displayed from the Write and WriteLn procedures. For example, run the following test program and True or False will be echoed back to the Text Window.

```
Program Boole1;
Var
   P : boolean;                 { George's Great idea }
Begin
   ReadLn (P);                  { Enter True or False }
   WriteLn (P)                  { Echo }
End.
```

Boolean variables are used to represent conditions in a Pascal program. For example, marital status, sex, and whether or not a person can drive a car can all be represented in a computer program using a Boolean variable for each possible condition.

```
Var
     MARRIED,                   { True means married }
     MALE,                      { True means male }
     Driver : boolean;          { True means can drive }
```

Boolean variables are used in a number of ways: (1) in expressions; (2) to determine which group of statements are to be executed (and consequently which ones are to be skipped over); and (3) to control the execution of loops. We will discuss the first two uses in this session and address the last use in the next session.

Boolean Operators

A *Boolean operator* specifies an operation which may be performed on Boolean variables and constants. Boolean operators may not be used with non-Boolean typed variables except when a Boolean result is obtained. As you might expect, there are only three Boolean operators: NOT, OR, and AND (see Table 6.1).

These operators are quite simple, but don't let them surprise you. The NOT operator works on a single Boolean variable. Not P is False if P is True, and True if P is False.

OR and AND work on two Boolean variables or constants at a time. Both P and Q must be False for "P or Q" to be False; both P and Q must be True for "P and Q" to be True.

TABLE 6.1 BOOLEAN OPERATORS

Boolean Operator	*Example*	*Explanation*
NOT	not P	Compute True only if P is false
OR	P or Q	Compute True if either P or Q is true
AND	P and Q	Compute True only if P and Q are true.

Explore these operators more fully by trying the following experiment. Enter this program into your Macintosh and RUN-GO.

```
Program Boole2;
Var
   P, Q : boolean;
Begin
   ReadLn (P, Q) ;
   WriteLn (P, ' AND ', Q ' =     P and Q ) ;
   WriteLn (P, ' OR ' Q, ' = ', P or Q) ;
   WriteLn ('NOT ', P , ' = ', not P)
End.
```

Watch what happens as you enter the pairs of inputs below (RUN-GO after each of the four experiments):

1. True True
2. True False
3. False False
4. False True

The outputs can be put into a *truth table* as shown. The first value (P) is listed horizontally and the second value (Q) is listed vertically. The result is listed in the table.

AND Truth Table

	P =	True	False
Q =	True	True	False
	False	False	False

OR Truth Table

	P =	True	False
Q =	True	True	True
	False	True	False

NOT Truth Table

P =	True	False
NOT P =	False	True

These two tables should agree with the results obtained from the example above. It will be important to memorize the AND, OR, and NOT tables for later use.

Boolean Expressions

While there are only three Boolean operators in Pascal, there are many more ways to obtain a Boolean result. Anytime two objects are compared, a True or False result may be obtained. The outcome of a comparison is a Boolean value.

A Boolean expression is an expression which evaluates to a Boolean value during program execution. A Boolean value may be obtained any time the operations shown in Table 6.2 are performed in Pascal.

The results shown in Table 6.2 are False if any comparison is not true. Thus, A = B is False if A does not equal B. A and B may be integers, real, or other types not yet discussed, but the result of comparing A and B yields a Boolean constant.

A Boolean expression may be used on the right-hand side of an assignment statement. For example, if P is a Boolean variable, then P may be assigned the result of the comparisons shown in the table. Here are some examples.

```
P := A = B;         { P is True if A equals B }
P := A <= B;        { P is False if A > B }
P := not P;         { P is changed to (not P)  }
```

Notice that A < = B is legal whereas A = < B is not. The < = and > = signs must be written as shown; they cannot be written in reversed order.

Boolean expressions can be combined with other Boolean operators if you are careful to use parentheses. The compound Boolean expressions below are legal and unambiguous.

```
P := (A = B)  and  (A <> 0) ;
P := not ( (A = B)  and  ( (A > 0)  or  (A < 100) ) ;
```

Sometimes you can do away with the parentheses, but to be safe use them—they cannot hurt.

Try the following program and observe what happens to the value of P as different values of A and B are entered.

TABLE 6.2 OPERATIONS RESULTING IN BOOLEAN VALUES

Operator	*Example*	*Explanation*
=	A = B	True if A equals B
<>	A<>B	True if A not equal to B
<	A<B	True if A less than B
>	A>B	True if A greater than B
<=	A<=B	True if A less than or equal to B
>=	A>=B	True if A greater than or equal to B
in	A in S	True if A is a member of set S

```
Program Boole3;
Var
    A, B : Integer;          { inputs }
    P    : Boolean;          { calculated output }
Begin
    Write ('Enter A, B ') ;
    ReadLn (A,B) ;
    P := not ( (A > B) or (B * A < 59) ) ;
    WriteLn (P)
End.
```

When 10 and 5 are entered, the output (value of P) is False, but when 6 and 10 are entered, the output is True. How can this program be analyzed to discover exactly what is happening? The answer is to use a truth table to clarify the expression.

```
P := Not ( ( A > B) or (B * A < 59) ) ;
```

Truth Table For P

inputs	(A > B)	(B * A < 59)	or	not
A,B = 10, 5	True	True	True	False
A,B = 6, 10	False	False	False	True

The truth table shows the Boolean value obtained at each step of the evaluation. First, (A>B) is evaluated, followed by (B * A<59). ThenOR is computed between the two subexpressions. The result is inverted by the Not operator to obtain a value for P.

Try running this program on your Macintosh. Remember to enter 5 and 10 on the same line separated by at least one space. Alternately, 5 can be entered on one line, followed by RETURN, and then 10 entered on the next line, followed by RETURN.

The truth table format can be used in many instances to clarify a program. The idea is simple, but because it is tabular, it makes comprehending a program much easier.

IF Statements

Computers are powerful tools for making decisions because programs can be written to take different actions depending on their input values. A computer-controlled automobile is much more versatile if it can decide which road to travel down on its own.

Programs decide which of several alternative actions to perform by evaluating a Boolean expression and then selecting one instruction or another, but not both. This is done using the IF statement in Pascal. Its form is shown in Figure 6.1. IF, THEN, and ELSE are reserved keywords used to punctuate this decision-making statement in Pascal.

Here is an example of an IF statement in Pascal:

```
IF P THEN
   WriteLn ('P is True')
ELSE
   WriteLn ('P is False') ;
```

Notice how IF P THEN is written on one line followed by the other part of the statement. Macintosh Pascal will automatically adjust an IF statement, thus forcing this format onto the statement. (Try entering it as one long sentence, all on the same line. As soon as you enter the semicolon, Macintosh Pascal rewrites it in this format.)

P is a Boolean variable (True or False), so when this statement is executed the decision whether to write 'P is True' or 'P is False' depends on the value of P at this point in the program. If P is True the THEN clause is performed; otherwise the ELSE clause is performed.

Enter the following program into the Macintosh Pascal Program Window and RUN-GO.

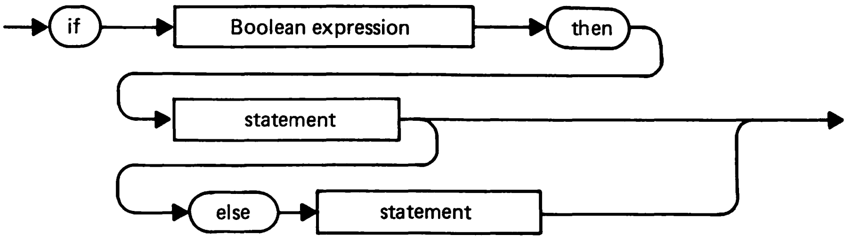


FIGURE 6.1
The format of an IF statement.

```
Program Boole4;
Var
    P : Boolean;            { input }
Begin
    ReadLn (P) ;
    If P Then
        WriteLn ('P is True')
    Else
        WriteLn ('P is False')
End.
```

Of course, if you enter "True," the result is 'P is True'; if you enter "False," the response is 'P is False'.

RUN-STEP

Now try the following alternate method of running this program. Pull down the RUN menu and select STEP instead of GO. Notice the pointing finger just to the left of the Begin keyword.

Select RUN-STEP a second time. The pointing finger will move to the ReadLn (P) statement and wait there until you type in "True" or "False." Enter "False" followed by return, and watch the pointing finger move to the IF statement.

Watch very closely now as you RUN-STEP one more time. The pointing finger will skip over the THEN clause entirely and point to the ELSE clause (which contains "WriteLn ('P is False')").

Do RUN-STEP one more time and watch the Text Window. The message 'P is False' appears in the Text Window and the pointing finger moves to the End keyword. Your program has stepped through each statement, one at a time.

RUN-STEP is a very useful tool for debugging programs containing IF statements. Since an IF statement introduces a certain element of uncertainty into programming, you will often wonder which way a program has proceeded. RUN-STEP makes quite clear which branch of an IF statement is actually done, and most importantly, which branch is ignored.

Putting in STOPS

In a large program it may be tedious to work your way through the program using RUN-STEP. Instead, you can place STOPS anywhere desired using the STOPS-IN option under RUN.

Select RUN-STOPS IN and then examine the left-hand margin that appears in the Program Window. At the lower left corner a small stop sign appears, indicating the availability of STOPS.

Move the mouse to the left-hand margin above the stop sign. The cursor will change into a stop sign when the cursor crosses over into the left margin. Move the stop sign next to the statement where you want the program to stop. Now click the mouse, and a stop sign will be fixed there. The stop sign marks a statement as the next statement to be executed after the program is resumed. (You can go on inserting as many stop signs as you like by repeating these steps.) Try this on Boole4 by moving a stop sign next to the IF statement.

Select RUN-GO and watch what happens. The program stops right before the IF statement is to be done. Pull down RUN-GO a second time, and the program resumes. This method should be used when you are not sure what a certain program is doing but you suspect a particular statement is creating a problem. Place a stop sign before and after the suspicious statement and RUN-GO several times.

To remove the stop signs, select RUN-STOPS OUT. Be sure to make the Program Window active before attempting to select RUN-STOPS OUT, since the option will be dimmed if another window is active.

One-Legged IF Statements

If you follow the lines in the diagram of Figure 6.1 it is possible to ignore the ELSE clause entirely. The line splits into two paths after the THEN clause. The ELSE clause is optional, as shown in the next example.

```
IF P THEN
   WriteLn ('To Write or not to Write . . .') ;
```

This statement tests the value of P, and if P is True, the message 'To Write or not to Write . . . ' is displayed in the Text Window. If P is False, the WriteLn statement is skipped and the IF statement has no effect on the Text Window.

Compound IF Clause

The IF statement chooses between two statements: Either one or the other is done, but not both. The THEN and ELSE clauses may be compound statements enclosed in Begin—End keywords, as shown in the next example. This greatly increases the power of IF statements because large portions of your program can be enclosed in either THEN or ELSE branches.

```
IF AGE > 21 THEN
   Begin
         WriteLn ('Greater than 21') ;
         AGE  :=  AGE  +  1           {optional semicolon here}
   End                                {no semicolon here}
```

```
ELSE
   Begin
        WriteLn ('Less than 22') ;
        AGE  :=  AGE  + 2
   End;                              {may not need semicolon here}
```

This example contains a compound statement in both branches of the IF statement. Be careful where you place semicolons in an IF-THEN-ELSE statement—*do not* put a semicolon before the ELSE keyword.

Hands-On Example of Compound IF Statements

Enter the program shown in Figure 6.2 into Macintosh Pascal and select RUN-GO. This program solves the quadratic equation

$$Ax^2 + Bx + C = 0$$

for the values of *x* (roots) which satisfy the equation. Given A, B, and C, there may be zero, one, or two solutions, and each solution may be either real or imaginary. (Entering the values: 0 0 1 causes the program to fail. Why?)

When the prompt 'Enter A, B, C : ' appears in the Text Window enter values: 1 2 1. Then press the return key. The following should appear:

```
REAL ROOTS
TWO ROOTS  =     -1.00          -1.00
```

Now select RUN-GO a second time and enter the three values: 1 1 1.

```
IMAGINARY ROOTS
TWO ROOTS  =     0.37          -1.37
```

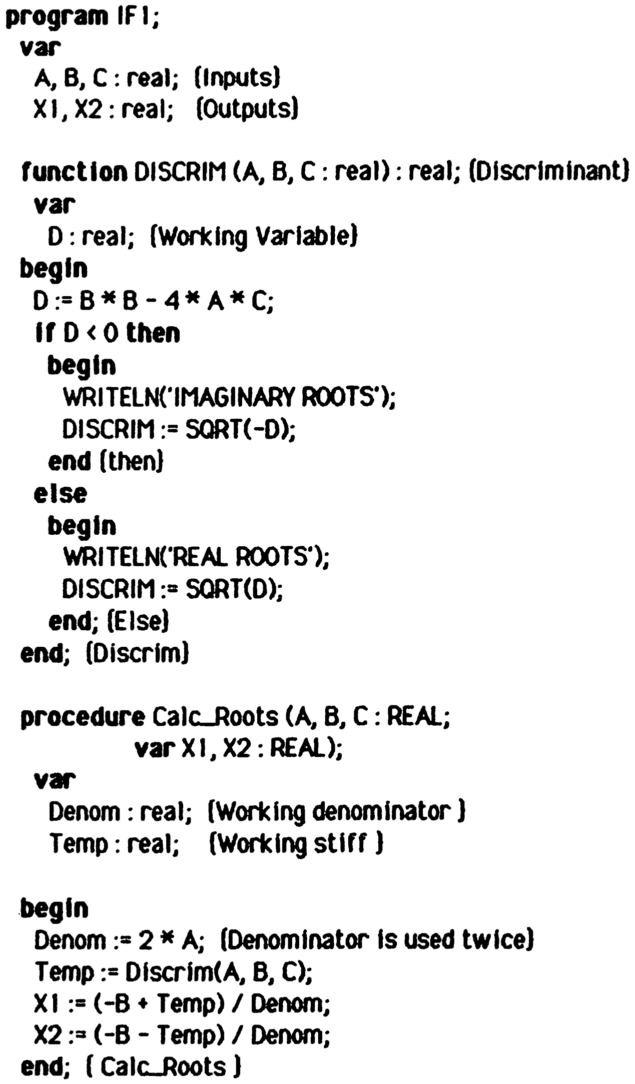
Finally, run the program and enter the three values: 0 2 1.

```
ONE ROOT          =  -0.50
```

The mathematical properties of this program are not important in this discussion. Instead, notice that this program illustrates a typical use of IF statements to perform some statements and avoid others. For example, in the body of the main program an IF A = 0 statement is used to decide whether to use the formula for a simple root or two roots. Without the IF statement the program would not work correctly for the case when A = 0.

The code for calculating the roots of a quadratic equation has a bug in it. It fails when A = 0 and B = 0 because of an attempted division by zero. The bug can be removed by inserting another IF statement inside of the IF A = 0 THEN clause. Here is the modified section.

```
IF A = 0 THEN
   Begin
          IF B = 0 THEN
               WriteLn  ('No Solutions')
          ELSE
               Begin
                     X1 := (-C/B) ;
                     WriteLn  ('One Root =',  X1:8:2)
               End           {Inner Then}
   End           {Outer Then}
ELSE
```



```
program IF1;
 var
  A, B, C : real;  {Inputs}
  X1, X2 : real;   {Outputs}

 function DISCRIM (A, B, C : real) : real; {Discriminant}
  var
   D : real;  {Working Variable}
 begin
  D := B * B - 4 * A * C;
  if D < 0 then
   begin
    WRITELN('IMAGINARY ROOTS');
    DISCRIM := SQRT(-D);
   end {then}
  else
   begin
    WRITELN('REAL ROOTS');
    DISCRIM := SQRT(D);
   end; {Else}
 end;  {Discrim}

 procedure Calc_Roots (A, B, C : REAL;
          var X1, X2 : REAL);
  var
   Denom : real;  {Working denominator }
   Temp : real;    {Working stiff }

 begin
  Denom := 2 * A;  {Denominator is used twice}
  Temp := Discrim(A, B, C);
  X1 := (-B + Temp) / Denom;
  X2 := (-B - Temp) / Denom;
 end;  { Calc_Roots }
```

FIGURE 6.2

Program IF1 solves $AX^2 + BX = C + O$.

```
begin {Body}
 Write('Enter A,B,C: ');
 Readln(A, B, C);  {Get AX^2 + Bx + C = 0}
 if A = 0 then
  begin
   X1 := (-C / B);
   Writeln('One Root =', X1 : 8 : 2);
  end {then}
  else
   begin
    Calc_Roots(A, B, C, X1, X2);
    Writeln('Two roots =', X1 : 8 : 2, X2 : 8 : 2);
   end
 end.
```

***FIGURE 6.2** (continued)*

This example illustrates a very subtle difficulty in Pascal. The IF B = 0 THEN statement is *nested* within the IF A = 0 THEN statement. One IF statement is inside of another IF statement. How does Pascal know which IF and ELSE clause to match up? The rule is easy to remember if you keep in mind that the IF statements are nested. Hence, the innermost ELSE clause is matched with the innermost IF statement, and so on.

If an inner-nested IF statement does not have an ELSE clause, then the rule for nested IF-THEN statements must be strictly obeyed. Remember that a "statement" can be another IF statement, and so on, up to any level of nesting.

Nested IFs: One IF statement may be nested within another IF statement for as many levels as desired. The rule concerning matching ELSE clauses with IF statements must be strictly obeyed. Innermost ELSE clauses go with innermost IFs, and so on, until all levels of IFs have been paired with their ELSE clause.

Multiway Branching

Nested IF statements become a necessary evil whenever your program must choose between three or more alternate paths. For example, suppose you want to write a procedure which categorizes people according to the first letter of their last names. All people whose last name starts with the letters A through E are placed in one category, F through J in a second category, and so forth, according to the table below.

Last	*Category*
A - E	'Tigers'
F - J	'Lions'
K - O	'Panthers'
P - T	'Leopards'
U - Z	'Sabres'

This problem can be solved in several different ways. The most general solution is to use nested IF statements as shown in Figure 6.3. The nested IFs are each tested, one at a time, as follows (RUN-STEP the program and watch what happens):

```
(Last >= 'A')  and  (Last <= 'E')
(Last >= 'F')  and  (Last <= 'J')
                "
                "
                "
(Last >= 'U')  and  (Last <= 'Z')
```

When a Boolean expression evaluates to True, the corresponding THEN clause is executed. After the corresponding THEN clause is executed, control skips over the remaining IFs. The outermost IF statement is terminated as soon as one of the Boolean expressions evaluates to True. This is clearly shown by the pointing finger when the program of Figure 6.3 is RUN-STEP for Last = 'Q'.

Case Statement

A better method of multiway branching may be possible if the values being tested are simple ordinal types (integer, Boolean, char) and the test conditions are uncomplicated. The case statement format is shown in Figure 6.4

The problem of categorizing people by the first letter of their last name can be solved using a case statement as shown in Program IF3 of Figure 6.5. RUN-STEP this program with Last = 'M'. Notice how the pointing finger skips over all case clauses except the one containing 'M' as a constant.

The case expression (Last) is called a *selector*, and the clauses are called *case clauses*. Each clause may contain a single statement, but of course, that statement can be a compound Begin–End statement made of additional nested statements.

```
program IF2;
 var
  Last : char;  {Input}
begin
 Write('Enter Letter: ');
 Readln(Last);
 if (Last >= 'A') and (Last <= 'E') then
  writeln('Tigers')
 else if (Last >= 'F') and (Last <= 'J') then
  writeln('Lions')
 else if (Last >= 'K') and (Last <= 'O') then
  writeln('Panthers')
 else if (Last >= 'P') and (Last <= 'T') then
  writeln('Leopards')
 else if (Last >= 'U') and (Last <= 'Z') then
  writeln('Sabres')
 else
  writeln('Error in Input')
end.
```

FIGURE 6.3
Program IF2 has six-way branching.

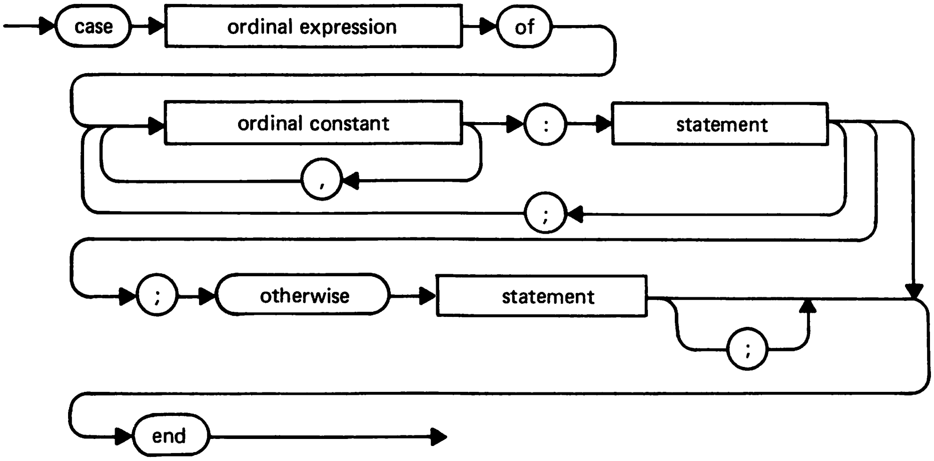
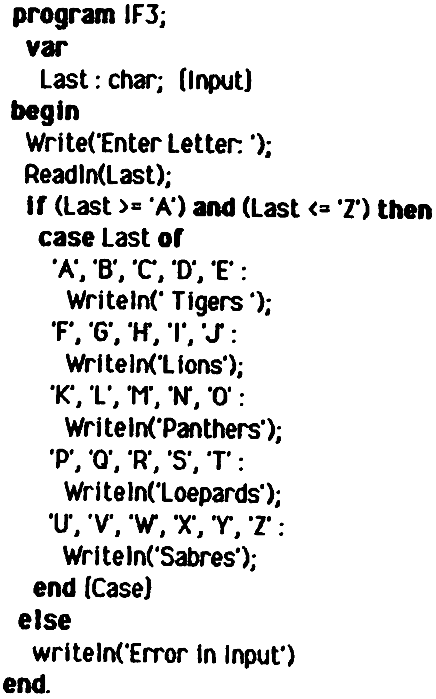


FIGURE 6.4
Format of case statement.



```
program IF3;
  var
    Last : char;  {Input}
begin
  Write('Enter Letter: ');
  Readln(Last);
  if (Last >= 'A') and (Last <= 'Z') then
    case Last of
      'A', 'B', 'C', 'D', 'E' :
        Writeln(' Tigers ');
      'F', 'G', 'H', 'I', 'J' :
        Writeln('Lions');
      'K', 'L', 'M', 'N', 'O' :
        Writeln('Panthers');
      'P', 'Q', 'R', 'S', 'T' :
        Writeln('Loepards');
      'U', 'V', 'W', 'X', 'Y', 'Z' :
        Writeln('Sabres');
    end {Case}
  else
    writeln('Error in Input')
end.
```

FIGURE 6.5
Program IF3 has a five-way case statement.

If the selection in Figure 6.5 fails to match one of the constants, an error occurs. This is why Program IF3 of Figure 6.5 has an IF statement around the case statement. The IF statement diverts bad inputs away from the case statement.

Suppose the IF statement was removed from the program in Figure 6.5. To do this, delete the IF-THEN and corresponding ELSE WriteLn ( ) statements. Now, RUN-STEP the program with lowercase 'a' as input.

This shows what happens when there is no corresponding case clause to match the selector. A thumbs-down sign indicates approximately where the search for a matching constant fails. Macintosh Pascal has an extended case statement as shown in the Figure 6.4 diagram. When an OTHERWISE clause exists and no matching clause is found while the Case is being executed, the OTHERWISE clause is executed.

Case statements have limited utility in Pascal programs, but when it is possible to use them, they increase clarity and simplicity. When it is not possible to use a case statement, use the nested IF-THEN-ELSE construction discussed earlier.

Case Statements: A case statement consists of a selector and one or more clauses to be selected. If no match is located among the clauses (and there is no OTHERWISE clause), the case statement is undefined and an error occurs. Every clause must be uniquely identified with a constant, or a list of constants separated by commas, or else the OTHERWISE clause must be included once. Only integer, char, and Boolean constants are allowed (exclude Real); therefore only integer, char, Boolean, and non-real type selector expressions are allowed.

A Hands-On Example

Enter the program of Figure 6.5 into Macintosh Pascal and RUN-STEP it with the following inputs.

Input	*Result*
A	Tigers
Z	Sabres
a	Error in Input

Why do you suppose the 'Error in Input' message occurred? How would you change the program so that upper- and lower-case letters are treated alike?

Modify the first clause so that it looks as follows:

```
'a', 'A', 'B', 'C' , 'D' , 'E' :
```

Now change the IF statement by replacing 'Z' with 'z'. RUN-GO the modified program. Does it display 'Tigers'? This experiment should suggest how you might modify the program to handle upper-case and lower-case letters alike.

Now change the first clause so that 'a' is 'aa' instead. RUN-GO the program and observe the error message.

```
TYPE UNKNOWN OR NOT IMPLEMENTED
```

You will most likely see this error message frequently when first using the case statement. A subtle error that you should remember occurs whenever a blank space is accidentally inserted between the letter and the single quotation mark: 'a'. Check all constants to make sure there are no embedded spaces.

The Observe Window

Now pull down the WINDOWS menu and select OBSERVE. The Observe Window will appear with the prompt:

```
Enter an expression
```

The blinking cursor indicates that you can enter any expression you want to watch while your program executes. Enter Last and press RETURN.

Select RUN-STEP and watch the value of Last in the Observe Window take on the value entered at the keyboard. After several RUN-STEPs enter the two letters PU followed by RETURN. Last becomes 'P' (the "U" is discarded).

RUN-STEP through the remainder of the program. The value of Last remains the same all the way through the program. Finally, Last becomes undefined when the program terminates.

Load one of the earlier programs and select WINDOW-OBSERVE. Watch the values of selected variables change as you RUN-STEP through the program. This technique is a very valuable method of locating bugs in new programs.

Summary

This has been a rather long session devoted to a difficult part of programming. Branches and Boolean logic account for five times as many errors in programs as other statement types. You should be very careful about how you use Boolean variables and twice as careful about IF statements.

It will take practice and frequent use of the Observe Window before you will become comfortable with IF statements. Fortunately, you will get a lot of practice in subsequent sessions.

Problem Solving

1. Modify the program of Figure 6.5 so that it recognizes upper- and lower-case letters of the alphabet.
2. Use a Begin-End compound statement to add the following WriteLn between the IF and case statements of Figure 6.5

```
WriteLn ('I am looking . . .') ;
```

What does this say about the IF statement format?

3. A clever way to change a lower-case letter into an upper-case letter is to subtract 32 from its ASCII code. This is done as follows for Last : char; .

```
If Last >= 'Z' Then
  Last := Chr (Ord(Last)  —  32) ;
```

Use this trick to improve the program in Figure 6.5.

4. Rewrite the program of Figure 6.3 to make it a procedure. Now write a main program which uses this procedure to categorize a person's first and last initials. Your program must call the procedure twice: once for each initial.

5. Write a program that does the following given the value of input variable NUMBER : integer; .

NUMBER	*Action*
1 or 2	WriteLn ('No way!')
3	Add 1 to NUMBER and show it
5	Add 1 to NUMBER and show it
7	Subtract 1 from NUMBER and show it
11	Subtract 1 from NUMBER and show it
13	Add 1 to NUMBER and show it

All other numbers cause an error message, 'Wrong Input,' to be displayed.

6. Write a program that does the following given the value of input variable NUMBER : Real.

NUMBER	*Action*
1.414	WriteLn ('Square root of 2')
between 0 and 1	WriteLn ('Zero')
between 1.415 and 2.25	WriteLn ('Greater than two')

Explain why Case cannot be used to solve this problem.

7. One tick of a clock means that one second has elapsed. Write a procedure called Tick (H,M,S) which adds one to S (seconds), and updates M (minutes) and H (hours) whenever the minutes and hours change.

Your procedure returns H, M, and S; hence, these must be declared as Var parameters.

The main program will accept H, M, and S as inputs; call Tick (H,M,S); and then output H,M,S.

Your code will look something like the following:

```
S := S + 1;            {one second tick}
If S > 59 Then
     Begin
           S := 0;
           M := M + 1;
           If M > 59 then
               Begin
                     M := 0;
                     H : = H + 1
                     If H > 23 Then             {. . .etc. . .}
```

Session 7:

Iteration Using While, Repeat, and For

In this session you will learn how to write sections of Pascal programs that are repeated. A statement is said to be in a loop *if it is executed repeatedly. Pascal has three kinds of looping statements: While, Repeat, and For. All three will be examined in this session.*

Algorithms

Muhammad al-Khwarizmi (780-850 A.D.) was an Arab mathematician whose writings are perhaps most responsible for converting the Western world from the cumbersome Roman numeral system to the efficient decimal system we use today. Not only is he regarded as the father of algebra, but the word *algorithm* is derived from his name, as well.

An algorithm is a mechanical procedure for doing a calculation. Computer scientists study what can and cannot be computed by an algorithmic process. They have discovered that everything which is computable can be computed by means of three actions: sequence, choice, and iteration. An algorithm is, simply stated, a recipe composed of three ingredients: sequence, choices, and iterations.

Iteration

You have already been introduced to sequences and choices. In this session you will learn how to use iteration in a Pascal algorithm. *Iteration* is the process of repeating one or more actions. It is sometimes called *looping* by programmers to refer to the process of branching back to an earlier statement within the program.

A loop has two components: a *body of statements* which are repeated and a *condition* which determines when the loop is to terminate. When a loop terminates, the statement immediately following the loop body is executed. The *loop condition* is a Boolean expression or *loop counter* which causes the loop to terminate at the proper time.

Of course, it is possible to accidentally write a looping program which never terminates. This is called an *infinite loop,* and it should be avoided (infinite loops terminate at the end of forever).

While Loops

The simplest loop in Pascal is called the *While* loop because it is implemented by writing a While statement as shown in Figure 7.1 and below:

```
While P Do
  S;
```

where:

P is a Boolean expression which evaluates to either True or False (the condition part).

S is a simple or compound statement (the body) to be iterated. A compound statement S may contain any other statements between a Begin-End pair, assignment, if, and even other While statements.

The While loop works as follows:

1. The Boolean expression P is evaluated.
2. If P is False, the body is skipped and the program goes on to the next statement following the body of the loop.
3. If P is True the body is executed.
4. After the body has executed, the While loop is executed again, starting with step 1, above.

Suppose, for example, you want to total 10 numbers as they are entered from the keyboard. Assume "SUM", "NUM" and "I" are integers and the following section of Pascal code is properly embedded within a complete Pascal program:

```
I     := 0;                     {loop counter and sum}
SUM := 0;                       {initialized, equal to zero}
While (I <= 10)  DO             {check loop condition}
 begin
    ReadLn (NUM) ;              {keyboard entry}
    SUM := SUM + NUM;           {add to running total}
    I      := I + 1             {count how many times}
 end;           {while}
```

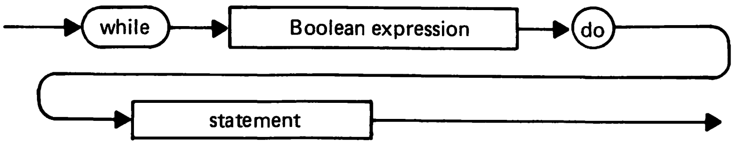


FIGURE 7.1
While loop syntax diagram.

This program section illustrates several important points:

1. The loop termination condition is a Boolean expression. When I < = 10 is not True, the loop will terminate.
2. The loop body is a compound Begin-End statement, which in turn contains three other statements.
3. The variables "I" and "SUM" must be initialized before the While loop and incremented during the loop (in the body). In the case of variable "I", the loop condition is changed each time "I" is incremented. "I" is used to test the loop termination condition *before* the loop body is executed. It is possible for the body to *not be executed at all* since the condition part is tested first.
4. The value of "I" corresponds to the number of times the loop body has been executed. Hence, "I" is sometimes called the *loop counter.*

Using a While loop requires you to force a certain condition to be False in order to terminate the loop. For example, a While loop was used to finally force (I < = 10) to be False, terminating the iteration, in the example above.

It is awkward to think in terms of negatives such as "when (I < = 10) not True", so another form of loop is available in Pascal. The Repeat-Until loop (conversely) forces the termination condition to be True.

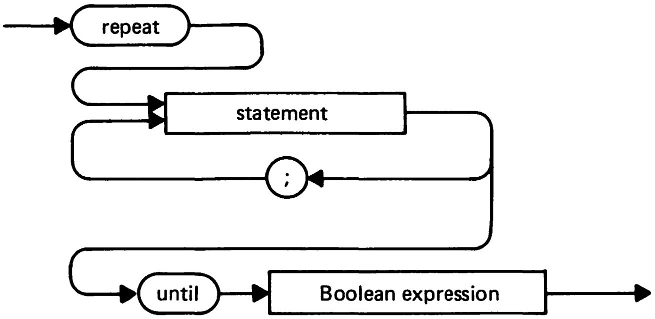


FIGURE 7.2
Repeat loop syntax diagram.

Repeat Loops

A *Repeat–Until* loop consists of a loop and a loop condition just like the While loop, but the loop condition is tested at the *end* of the body instead (see Figure 7.2). This means that the loop body is *always* executed at least once.

```
Repeat
    S1;
    S2;
     "
     "
     "
    Sn
Until P;
```

You should notice immediately the differences between While and Repeat loops. First, the Repeat loop body may contain one or more statements without resorting to a nested Begin-End compound statement. The statements S1; S2; . . . Sn may be assignment, While, Repeat, If, or any valid Pascal statement.

The second big difference between a Repeat and While loop is the placement of the loop condition. The Boolean expression P is tested *after* the body is terminated only when P is True. Here are steps in executing a Repeat–Until loop.

1. Execute the body of the loop. This includes all statements between the reserved keywords Repeat and Until.
2. Test P; if it is False, repeat from step 1. If P is True, the loop terminates and control passes on to the next statement following the keyword Until.

Here is an example in which the loop condition forces a user to enter the proper value into a running program:

```
Repeat
      Write ('Enter a number between 0 and 10: ') ;
      ReadLn (NUM)
Until (NUM <= 0) and (NUM <= 10) ;
```

The only way the user can continue is to obey the prompt and enter a number between zero and ten. The loop condition becomes True when the entered value is "between zero and ten."

It is important to use parentheses as shown in this example, because Pascal cannot understand an ambiguous Boolean expression. Always use plenty of parentheses when writing potentially ambiguous expressions.

For Loops

In a few situations a third possibility arises where it is not convenient to use a loop condition to terminate the loop. Instead, a loop *counter* is preferred. The *For* loop is used in this case (see Figure 7.3). Notice that the keyword "To" may be replaced with "Downto," which is used in a loop that counts down from a larger starting value to a smaller stopping value.

```
For I := I_START To I_STOP Do
    S;
```

"I" must be an ordinal type; I_START and I_STOP must evaluate to an ordinal value. This means I, I_START, and I_STOP can be integers, characters, and scalars (to be discussed later).

Here is how the For loop works.

1. The loop counter "I" is initially assigned the value of I_START. I_START may be an expression; if it is, the expression is evaluated and then its value is assigned to "I".
2. The loop counter "I" is compared with the terminal value I_STOP. I_STOP could be an expression; if so, the expression is evaluated and compared with "I".
3. If "I" exceeds I_STOP, the loop body S is skipped and the statement following S is executed, instead.
4. If "I" does not exceed I_STOP, the loop body S is executed. S may be a single statement or a compound Begin-End statement.
5. The loop counter is incremented by 1 (To) or by -1 (Downto). Then, control goes back to the For statement and steps 2 through 5 are repeated.
6. "I", I_START, and I_STOP *must* be of the same type.
7. The value of "I" *must not* be changed or modified inside the loop body.
8. "I" is considered undefined outside of the loop.
9. If I_START is greater than I_STOP, the loop is not executed at all.

The following For loop and While loop statements do exactly the same thing:

```
For I := 1 To 10 Do            I := 1;
    S;                         While I <= 10 Do
                                   begin
                                       S;
                                       I := I + 1
                                   end;
```

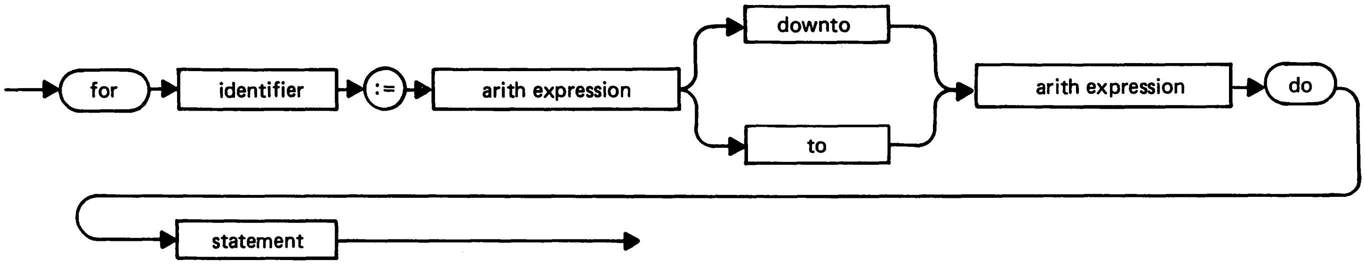


FIGURE 7.3
For loop syntax diagram.

Here is still another way to accomplish this exact same thing:

```
For I := 10 Downto 1 Do          I := 10;
    S;                           While I >= 1 Do
                                     begin
                                         S;
                                         I := I - 1
                                     end;
```

As you can see, the For loop is more direct and concise than the equivalent While loop in this case. The For loop is only appropriate when a loop counter is used, however. The For loop is not as general as a While or Repeat loop.

Hands-On Looping Example

Start up Macintosh Pascal and enter the procedure shown in Figure 7.4. This procedure can be executed by editing the following statements into the Program Window text:

1. Add a program heading:

```
Program MAIN;
```

2. Add AVG to the Declaration Part, following the header:

```
Var
   AVG : Real ;              {actual parameter}
```

3. Add a program body containing the call to CALC_AVG, as follows:

```
Begin                    {MAIN}
   CALC_AVG (AVG) ;
   WriteLn ('AVG      = ' , AVG :12:3)
End.                     {MAIN}
```

4. RUN-GO.

Try this example with a few numbers. The average value should appear in the Text Window. What happens when you try to enter a negative zero value for N?

Now select WINDOWS-OBSERVE and enter the name AVG into the first cell of the Observe Window. Keep this window and the Text Window visible while you RUN-STEP-STEP your program. You may want to close the Program Window to make room for everything on the screen. Now watch as AVG changes values. Does this explain what the procedure does?

Add the variable "I" to the next cell in the Observe Window. RUN-STEP-STEP again and watch what happens to the value of "I". Clearly, the Observe Window is a valuable debugging aid for "watching" loops while they execute.

```
procedure CALC_AVG (var AVG : real);
 var
  I, N : integer; {loop counter and upper limit}
  NUM : real; {input numbers to be averaged}
begin
 repeat
  Write('Enter number of numbers(N>0)');
  ReadLn(N);
 until (N > 0); {repeat}
 AVG := 0.0;
 for I := 1 to N do {N>0}
  begin
   Write('Enter a number');
   ReadLn(NUM);
   AVG := AVG + NUM {accumulate running total}
  end; {for}
 AVG := AVG / N; {N>0}
end; {CALC_AVG}
```

FIGURE 7.4
Procedure CALC_AVG.

Summary

The three loop statements in Pascal are While, Repeat-Until, and For. The For loop is limited to counting the number of times its body (compound statement) is repeated. The repeat statement to open always executes at least once, and the While statement is used when the test is to be done before the loop is entered.

You should use the Repeat and While loops to force a certain condition to be True or False. The condition then becomes the loop condition. The While loop condition must be able to eventually become False, and the Repeat loop condition must be able to eventually become True.

Problem Solving

1. Fire up your Macintosh Pascal and enter the following program fragments (you will have to add other statements to complete the program in each case). What do these loops do?

a.
```
P := 1.0;
For I := 1 To 5 Do
    P := P * I;
```

b.
```
For I := 1 To 39 Do
    WriteLn ( 2 * I, 2 * I — 1 ) ;
```

c.
```
For I := 1 To 10 Do
    WriteLn ( I * I ) ;
```

d.
```
Repeat
     ReadLn ( Ch ) ;                {Ch: char}
     WriteLn ( Ord (Ch) — Ord ('A') )
Until Ch =' ';                 {blank}
```

e.
```
While X <> Y DO            {X, Y: Integer}
     If X < Y Then
       Y := Y — X
     Else
       X := X — Y
```

2. Under what conditions (values of A and B) will this loop never terminate?

```
While A <> B DO
     begin
          A := A + 1;
          B := B - 1
     end;             {While}
```

3. Under what conditions (values of A and B) will this loop never terminate?

```
Repeat
          A := A + 1
Until A > B;
```

4. Write a procedure called RAISE which raises a real number X to a power of P where P is an integer. The procedure heading is:

```
Procedure RAISE (X : Real; P : Integer; Var RESULT : Real) ;
```

5. Write a procedure called GUESS containing the constant ANSWER = 67, which does the following:

   a. Loops until the user correctly guesses the value of ANSWER.
   b. Prompts the user to enter an integer.
   c. Tells the user to guess again (a higher or lower number) if the wrong value is entered.
   d. Tells the user how many guesses were required to correctly guess the value of ANSWER.

   Your procedure should have a While or Repeat loop with an appropriate loop termination condition in it. Incorporate GUESS in a program and try it on a friend.

Session 8:

The Type Statement

In this session you will learn how to create your own data types in Pascal and to put these new types in the data declaration part of a program, procedure, or function using the type statement.

The Notion of Data Types

Charles Babbage is known as the father of computing; in the mid-1800s he invented several mechanical "engines" capable of automatic operation under the guidance of a program punched into wooden cards. Babbage was an advanced thinker highly critical of the British scientific community. He spent many years of his life attempting to build the "analytic engine," but unfortunately it was never constructed.

The program of the "analytic engine" was to have been encoded in punched cards of different colors: white for addition, yellow for subtraction, blue for multiplication, and green for division. This scheme would allow Babbage to check the number of operations actually performed against the number of cards of each color.

Babbage had so many ideas which closely match the ideas used in modern computing that he is credited with being the first computer scientist even though his ideas were not put to work for 100 years. The idea of color-coding punched cards, for example, is similar to the concept of "types" in computing.

We have previously defined a type as a set of values. Now we can refine this notion since you have experienced several different data types and the legal operations permitted on them.

Type: The set of objects and all of the permissible operations that may be performed on objects of the given type.

The set of objects called integers is actually a set of values and a set of operations. For example, the integer operators div and mod only work with integers. Therefore, div, mod, +, -, =, ord, chr, and so forth are the only permissible operations on integers. Try writing a Pascal program with two real variables X and Y, and apply the mod operation on them.

```
X mod Y;
```

This will result in a bug dialog which tells you this operation cannot be performed on variables of this type.

On occasion, Pascal violates its own strict rules concerning mixing types and their operators. For example, integers I and J are coerced when the real division operator is applied to them.

```
I / J;
```

The advantage of this exception is that it allows you to be a bit careless, and it makes Macintosh Pascal take care of the details of type conversion. The disadvantage, of course, is the confusion and sometimes the errors that mixing types like this causes.

In general, relaxing strict typing in Pascal is acceptable and beneficial. But be careful not to be led into run-time errors when bending the rules.

The Type Statement

One of the most elegant features of Pascal is its ability to accept programmer-defined types. This is done with a *type statement,* which optionally goes after the Const statement, but before the Var statement. The form of a type statement is shown below (and in Figure 8.1):

```
Type
    id1 = type;
    id2 = type;
         "
         "
         "
    idn = type;
```

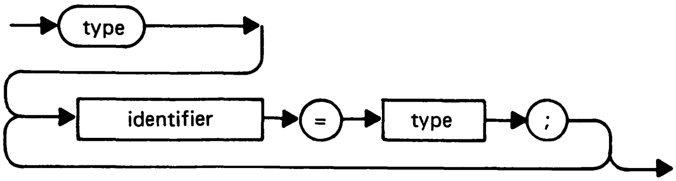


FIGURE 8.1
Type statement syntax diagram.

where:

id is an identifier which is the name of the type.
type is the set of values (and operators) associated with the id.

Here is an example of a type statement:

```
Type
     PAYMENTS       = Real;
     CODE_LETTER = Char ;
     AGE                = Integer;
```

The identifiers PAYMENTS, CODE_LETTER, and AGE are the names of types, *not variables.* AGE can be used to declare the type of a variable in a Var statement, for instance:

```
Var
      X, Y, Z, : AGE;          {new type}
```

As you can see, the new type called AGE more fully describes what X, Y, and Z really represent. While this enhances the readability of any Pascal program, the power and convenience of establishing your own types goes far beyond mere substitution for integers, real, and char.

Subrange Types

A *subrange* type is a subset of a type. Often it is a good idea to restrict the set of characters or integers to a smaller collection of values. When this is done, we say the new subset is a subrange of the base type. The values that a variable potentially represents belong to that variable's *base type.* The base type must be an *ordinal type* (a set whose members can be counted in whole numbers—integer, char, Boolean, and some user-defined types—reals are obviously excluded).

```
Type
     AGE     = 0 .. 99;        {restricted integer}
     WEIGHT = 0 .. 399         {restricted integer}
Var
     PERSON : AGE;             {type is applied here}
```

This illustrates how the type statement is used to create two new types: AGE and WEIGHT (each of which is based on the set of integers). They are restricted, however, to the integers from 0 to 99 (AGE) and 0 to 399 (WEIGHT).

A subrange is designated by the two periods (..) between its lower value and the upper value. Pascal will not allow any values outside these value limits to be stored in variables of a restricted type. To observe this, warm up your Macintosh Pascal and attempt the following hands-on exercises.

Hands-On Experiments with New Types

Enter the following miniature program and RUN-GO

```
Program MAIN;
  Const
     N = 10;
  Type
     I  = 0 .. N;            {constants allowed}
  Var
     I_TEST : I;          {new type}
  Begin
     I_TEST := 11;                    {won't work}
  End.                  {MAIN}
```

Notice what happens because I_TEST exceeds its allowed collection of values. This bug message may appear unexpectedly someday when you are running a debugged program. It shows how serious Pascal is about type checking.

A type is like a template or house plan which tells Pascal variables what values they can accept and what operations are permitted. To see the effect of changing the base type, modify the sample program as follows:

1. Replace the definition of "I" in the type statement with this new definition:

   ```
   I = Real;                {base type is real}
   ```

   This causes I_TEST to change from an integer to a real. Now RUN-GO and see what happens. Obviously, the 11 has been converted to 11.0 and stored in I_TEST.

2. Replace the entire type statement with the following more complex statement:

```
Type
      COLORS = Integer;
      SHADES = Char;
      HUES     = COLORS;
      I           = 0 .. N;              {same as before}
```

3. Add the following variables to the list of variables in the Var statement:

```
HUE     : HUES;
SHADE : SHADES;
```

4. Finally, add these executable statements to the body of the program and select RUN-GO.

```
HUE     := I + HUE;
SHADE := I;
```

   Here again, the first assignment statement works because the base types of I and HUE are the same, even though they are different types. The "+" operator works on integers regardless of their restrictions.

   The second assignment statement fails, however, because the base types are incompatible. SHADE is a character (char) and I is an integer.

Notice the transitive type relationship of HUE. HUE is of type HUES; HUES is of type COLORS; and COLORS is of type Integer. Any type can be defined in terms of previously defined types. The order of definitions is critically important, however, because a type must be defined before it can be used. Try to rearrange the types as follows and see what happens when you select RUN-GO.

```
Type
      HUE        = COLORS;
      COLORS = Interger;
      SHADES = Char;
      I            = 0 .. N;
```

As a final experiment, change the type declaration of SHADES to a restricted set of characters and observe the results.

```
SHADES = 'a' ..'z';                    {lower case only}
```

Characters can be restricted to a subset of all keyboard characters in the same manner as integers are restricted to a subset of all allowable whole numbers.

Examples of Subrange Types

The subrange type mechanism is very important in Pascal because it improves the readability, reliability, and maintainability of your programs. Because of these "abilities," we want to encourage you to use the subrange mechanism as much as possible. Here is a list of useful subrange types to give you some ideas of your own.

```
Type
     CARDS   = 1 .. 52;              {deck of playing cards}
     WEEKS   = 1 .. 52;              {weeks in a year}
     DAYS    = 1 .. 31;              {days in longest month}
     HOURS   = 1 .. 24;              {24 hours per day}
     MINUTES = 0 .. 59;              {60 min per hour}
     YEARS   = 1984 .. 2001;         {rest of century}
     INCHES  = 0 .. 11;              {12 inches per foot}
     FEET    = 0 .. 2;               {3 feet per yard}
     UPPERS  = 'A' .. 'Z';           {upper case letters}
     DIGITS  = 0 .. '9';             {numerals}
```

Remember that subrange types must be restricted to ordinals. Hence, you cannot restrict real values to a subrange. Both positive and negative bounds may be used; the operators which apply to the base type also apply to the subrange type.

Summary

A *type* is a set of values and a set of operators. The ordinal types may be restricted to subranges using the " .. " notation in either a type or a Var statement. For example,

```
Var
     CAT : 0 .. 2;
```

is valid, as well as,

```
Type
     CATS = 0 .. 2;
Var
     CAT : CATS;
```

The difference is that CATS becomes a reusable type in the second illustration and can be used in subsequent definitions.

Most types cannot be mixed in an expression (don't "add feet to inches"), but several exceptions are permitted in Pascal.

Integers may be coerced by the real operators " / " and " := ". Furthermore, integer values can be entered into real variables without harm using the ReadLn procedure. You should be careful about using mixed types, however, because they may be the source of bugs.

Macintosh Pascal Types

There are four standard ordinal types in Macintosh Pascal. Notice that LongInt is nonstandard.

integer	is -32,768..+32,767.
longint	is -2,147,483,648..+2,147,483,647.
char	is "any keyboard character."
Boolean	is False..True.

There are four real types in Macintosh Pascal. Extended, Double, and Computational are nonstandard.

real	is ±1.5e-45 to ±3.4e+38 (7 digits).
double	is ±5.0e-324 to ±1.7e+308 (15 digits).
extended	is ±1.9e-4951 to ±1.1e+4932 (19 digits).
computational	is a fixed-point real number where the exponent is always zero (19 digits).

In addition, see the description of the SANE library in Appendix D of the *Macintosh Pascal Reference Manual* from Apple Computer Corp.

There are six other *structured types* which will be discussed in later sessions.

array	is an object with many values of the same type.
set	is a set of ordinals.
file	is a disk file or device such as the printer.
record	is an object with many mixed-type values.
string	is textual data—an ordered sequence of chars.
pointer	is an address of one or more objects.

The type statement is very powerful when combined with more sophisticated data types. In future sessions, we will expand on this idea to build powerful data structures.

Problem Solving

1. Use the two types below in a procedure for adding two lengths together, measured in feet and inches.

```
Type
    FEET   = Integer;
    INCHES = 0 .. 11;
```

```
Var
    Feet1, Feet2 : Feets;
    In1, In2     : Inches;
```

(*Note:* Pascal doesn't differentiate between upper- and lower-case identifiers, so FEETS and Feets name the same object.)

Your procedure should accept Feet1, Feet2, In1, and In2 as inputs; calculate the SUM of Feet1, In1; Feet2, In2 (converted to feet and inches); and return the sum in variables Feet2 and In2.

[*Hint:* be wary of exceeding 11 inches since INCHES = 0 .. 11. This can be avoided by checking the sum before it is stored in a variable of type INCHES.]

2. A cubic yard of sand is 27 cubic feet. Let Cubic_Yds and Cubic_Fts be types; CYDS and CFTS be variables.

```
Type
    Cubic_Fts  = Real;
    Cubic_Yds  = Real;
Var
    CYDS : Cubic_Fts;
    CFTS : Cubic_Yds;
```

Write a procedure called CONVERT_TO_FT which converts cubic yards into cubic feet. Use the types and Vars given here.

3. Using the following Types, write a procedure called TIC_TOC that advances a hypothetical clock by one second of time each time it is called.

```
Type
    HOURS = 0 .. 23;
    MINS  = 0 .. 59;
    SECS  = 0 .. 59;
```

The procedure takes the current time as inputs and returns the current time, plus one second.

```
Procedure TIC_TOC (Var HR : HOURS;
  Var MIN : MINS; Var SEC : SECS);
```

4. Suggest types for the following:
   a. Test scores as a percent.
   b. Number of students in a class.
   c. Number of squares in a checkerboard.
   d. Number of months in a year.
   e. The years from 100 BC to 2000 AD.
   f. A person's initials (first and last).
   g. The amount of an employee's hourly pay.
   h. A person's weight to the nearest tenth of a pound.
   i. The number of suits in a deck of playing cards.
   j. The number of dots on a gambler's die.

Session 9:

Text Files and Printer Output

In this session you will learn how to write text to a disk file, read it back, and write text to the printer from a running Pascal program. In addition, you will learn about sequential files; how to open and close them, and how to use many of the elementary file functions and procedures built into Macintosh Pascal.

What Is a File?

Herman Hollerith was responsible for inventing the first machine for automatically processing huge quantities of information. In 1880 his tabulating machines were used to help count the number of people living in the United States, and later his company became International Business Machines. Because of Hollerith we have punched cards which are exactly the same size as a dollar bill, and 80-column printers (IBM discontinued using punched cards in 1984).

Hollerith's idea was simple, yet brilliant. Each person's census data was punched on an 80-column card. The millions of cards were stored in filing cabinet drawers so the census bureau could easily handle them, look up information, and store the data for long periods of time. This way of organizing large amounts of information is still used today in what we call data files.

Throughout this session you will be introduced to many new terms, but the notion of a data file is very simple. Think of a data file as a cabinet drawer full of cards. Each card is a file record, and the information punched into each column merely a single character per column.

Inside your Macintosh are programs for opening, closing, and reading through a file in a manner quite like opening, closing, and reading a drawer full of punched cards. The trick is to learn the names of the file system programs, how they are used in Macintosh Pascal, and then how to put them together so you can do useful file processing.

A *file* is simply a region on diskette which has a name and other information associated with it. (Use FILE-GET INFO to see this information.) Each file is divided into units of storage called *file records.* File records should not be confused with the record structure discussed in a later session.

Figure 9.1(a) illustrates the logical structure of a file. Records are numbered from zero to LAST; files can be up to 2,147,483,647 records in length. The last record is followed by a special end-of-file marker called EOF. A program cannot read beyond the EOF marker.

Kinds of Files

One way to classify Pascal files is by how they are processed. In a *sequential file,* records are processed in order, one after the other. Record zero is processed first, then record one, two, . . . until all records have been processed.

Alternately, a *random access file* processes records in an arbitrary (random) order. The second record may be processed, then the first, then the last, and so forth. Random files are sometimes called *direct files.* We will discuss only sequential file processing in this session.

There are two basic file structures in Pascal: text and typed. A *text file* may contain any kind of information (perhaps purely numerical information or a mixture of numbers and characters). We will discuss only text files in this session.

A text file must be processed sequentially; therefore a text file is also a sequential file. The records of a text file are strings of (possibly) varying length,

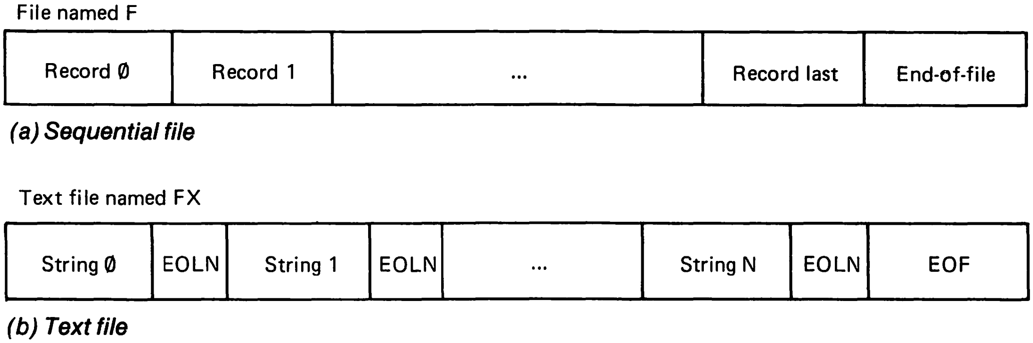


FIGURE 9.1
Logical structure of a file.

terminated by an *EOLN* (end-of-line) marker. The EOLN marker is the carriage return character produced by pressing the return key. The structure of a text file is shown in Figure 9.1(b).

Declaring a Text File

A text file is declared in the data declaration part by simply listing its internal name in the Var statement.

```
Var
      F : Text;            {file of strings}
```

This declaration causes Pascal to reserve main memory for a single record. Thus, when a file is read into main memory only one record at a time is copied from diskette to F. Similarly, when a file is written to a diskette only one record at a time is copied from F to the diskette.

There is a difference between the name of the file within a running Pascal program and the name used to refer to the file on the diskette. The diskette stores a list of filenames (and their icons) on the diskette itself, but Pascal keeps a different name within main memory while running a program. In this example, F is the name kept in main memory by the Pascal program. F is called the *internal file name,* and the name kept on the diskette is called the *external file name.* It is very important to know the difference between an internal and external file name.

Opening a File

A file must be *opened* before it can be read or written. An Open procedure associates the internal file name with the external diskette name. There are two Open procedures for sequential files:

reset Open an existing file for reading data into main memory.

rewrite Open a new file for writing from main memory to diskette; if the file already exists it is erased and new data are written over the old data.

Two parameters are needed to open a file using either one of these two procedures. The first parameter is the internal name of the file; the second is the external name (or string variable containing the name) given in the diskette directory.

Typically, a program must use a variable to hold the file records as they are copied in and out of main memory and another variable to hold the name of the diskette file.

```
Var
   Ftype : Text;             {internal name}
  Fname : String;            {external name}
```

Now, when the file is opened the program must provide the value of Fname, as shown below.

```
ReadLn (Fname);
Reset (Ftype, Fname);          {associate}
```

During program execution, the internal name Ftype is associated with the external diskette name in Fname.

Macintosh Pascal has an intrinsic function called *NewFileName (Prompt)* which returns the filename (obtained from a dialog box), that you may wish to use. In this case, your program gets the external filename from the dialog and associates it with the internal filename, as follows:

```
Fname := NewFileName ('Enter filename:');
Reset (Ftype, Fname) ;
```

Closing a File

After a file is used, it must be closed in order to copy out the last record processed and to append an EOF mark to the end of the diskette file. Remember that the file is processed one record at a time. If the last record is still in main memory, the Close procedure forces it to be written to the diskette. For example, Ftype can be closed by simply writing the following:

```
Close (Ftype);
```

Notice that only the internal file name is used as a parameter in the Close procedure.

Hands-On File Output

Suppose you want to enter a single line of text into a file called DATA. DATA is the external diskette directory name; Ftype is the internal filename used by the running program.

```
Var
   Ftype  : Text;              {output file}
   Line   : String;            {to be written}
   I      : Integer;           {used later}
   Fname  : String;            {used later}
```

The code to do this is given below.

```
Begin          {MAIN}
   Rewrite (Ftype, 'DATA');    {associate}
   ReadLn (Line);              {get from user}
   WriteLn (Ftype, Line);      {output it}
   Close (Ftype)               {close file}
End.           {MAIN}
```

The Rewrite procedure associates Ftype with 'DATA' so the program knows where to write the text to disk. Rewrite prepares the file for copying, one record at a time, from main memory to diskette.

WriteLn (Ftype, Line) looks like any other WriteLn procedure, but notice the first parameter. Ftype is the name of the internal file record containing the information to be written to 'DATA'. Here is what takes place when this procedure is executed.

1. The string in Line is copied to Ftype^.
2. The diskette file associated with Ftype is written to by copying the contents of Ftype^ to record zero of 'DATA'.
3. The next record (record one) is prepared to receive another string of text.

Notice the ^ following Ftype. This means to copy the information that Ftype points to rather than Ftype itself. (The ^ means "points to.") Ftype^ is the *file buffer* or area in main memory used to temporarily hold one record while the diskette is being accessed.

The Close procedure writes an EOF marker at the end of the file. The association between Ftype and 'DATA' is dropped and the file is closed.

Enter this small program into Macintosh Pascal and RUN-STEP-STEP. You will hear the diskette run, but otherwise nothing unusual will occur. This means the line of text you entered has been written to a disk file called DATA. If you FILE-QUIT and look at the file icons in the opened disk window, a new icon called DATA will appear there.

Replace the body of this program with the following statements. These statements take *N* lines of input from the keyboard and write them to a file called Fname. Each line of text is copied into Ftype^, then moved to a subsequent record of Fname.

```
Begin
   Rewrite (Ftype, Fname);          {associate}
   For I := 1 To 10 Do              {I : Integer}
 begin
   ReadLn (Line) ;                  {from keyboard}
   WriteLn (Ftype, Line)            {to disk file}
 end; (for)
Close (Ftype)                       {EOF marker}
End.
```

To make this program run, you must add a line at the top which gets a value for Fname.

```
Fname := NewFileName ('Enter filename:');
```

Now select RUN-STEP-STEP and enter DATA as the value of Fname. This causes the old DATA file to be removed and another to be rewritten over it.

Enter ten lines of text, each line terminated by EOLN (an end-of-line marker is generated by pressing the return key). The disk whirrs and copies each line to file DATA.

Reading a Text File

Suppose you want to read the text from file DATA back into the program and display it. This is done by reversing the process and using Reset in place of Rewrite.

```
Begin
   Reset (Ftype, Fname);              {associate input}
   For I := 1 To 10 Do
      begin
         ReadLn (Ftype, Line) ;       {get a record}
         WriteLn (Line)               {text window}
      end;
   Close (Ftype)                      {flush}
End.
```

This method works fine when you already know how many records are in the file (ten in this example), but most of the time this is unknown. So instead of a For loop, a While loop is used. The termination condition is given by an intrinsic function called EOF. EOF returns True if an EOF marker is read. The loop is better written as follows:

```
While Not EOF (Ftype) Do
      begin
        ReadLn (Ftype, Line) ;    {get one}
        WriteLn (Line)            {show it}
      End.
```

Each record, starting from record zero, is read and displayed. The EOF returns False until the last record is obtained. When the EOF marker is sensed by the EOF function, the loop terminates.

General Forms of I/O

Some general forms of input and output are summarized here for purposes of quick reference. For reading a text file use the following:

```
Reset (       );
While Not EOF (       )
   begin
       "
       "
       "
   ReadLn (       );
 end;
Close (       );
```

Simply fill in the blanks to get a working program body.

The form for writing a new text file is shown below; a Repeat loop is used to allow an arbitrary termination condition. This method allows you to enter any number of text lines without providing beforehand the number of lines to be entered.

```
Rewrite (       );
Repeat
     ReadLn (       );          {from keyboard, etc.}
     Done :=       ;            {some termination condition}
     If Not DONE Then
       begin
          WriteLn (       );    {write to file}
              "
              "
              "
       end          {If}
Until DONE;                     {termination ? }
Close (       );
```

The value of DONE is either True or False depending on whether or not more text is to be processed. For example, DONE might be True if the length of the string is zero, indicating that only a RETURN *(null string)* was entered, without any characters. Hence, when a null string is entered, no more text is processed.

```
     DONE := (Line = '');      {'' is a null string}
```

The Printer

Macintosh Pascal treats devices such as the printer and modem like text files. When a device looks like a file to a computer, it is said to be a *pseudo-file*. Macintosh Pascal "knows" two useful pseudo-files: PRINTER: and MODEM:.

Output from a running program can be directed to the printer by opening a pseudo-file called PRINTER:.

```
Var
     Printer : Text;                  {internal name of printer}
Begin
     Rewrite (Printer, 'PRINTER:');   {external name}
          "
          "
          "
     WriteLn (Printer, . . . );       {DO IT}
          "
          "
          "
     Close (Printer)
End.
```

Don't forget to include the colon at the end of "PRINTER:" when using it as a pseudo-file name. The internal name may be anything as long as the Rewrite associates it with the pseudo-file named PRINTER: .

Copy a File to the Printer

Try the following procedure for copying a text file from a diskette to a printer. This procedure is derived from the general form for file I/O, provided previously.

```
While Not EOF (Ftype) Do           {go to end of Ftype}
  begin
    ReadLn (Ftype, Line);          {get a line}
    WriteLn (Printer, Line)        {. . print it}
  end;
```

The complete procedure appears in Figure 9.2 as Procedure COPY_FILE_TO_PRINTER. Given that the name of the diskette file is Fname, this procedure opens both the diskette file and printer pseudo-file, reads one, and writes to the other, until all records have been processed.

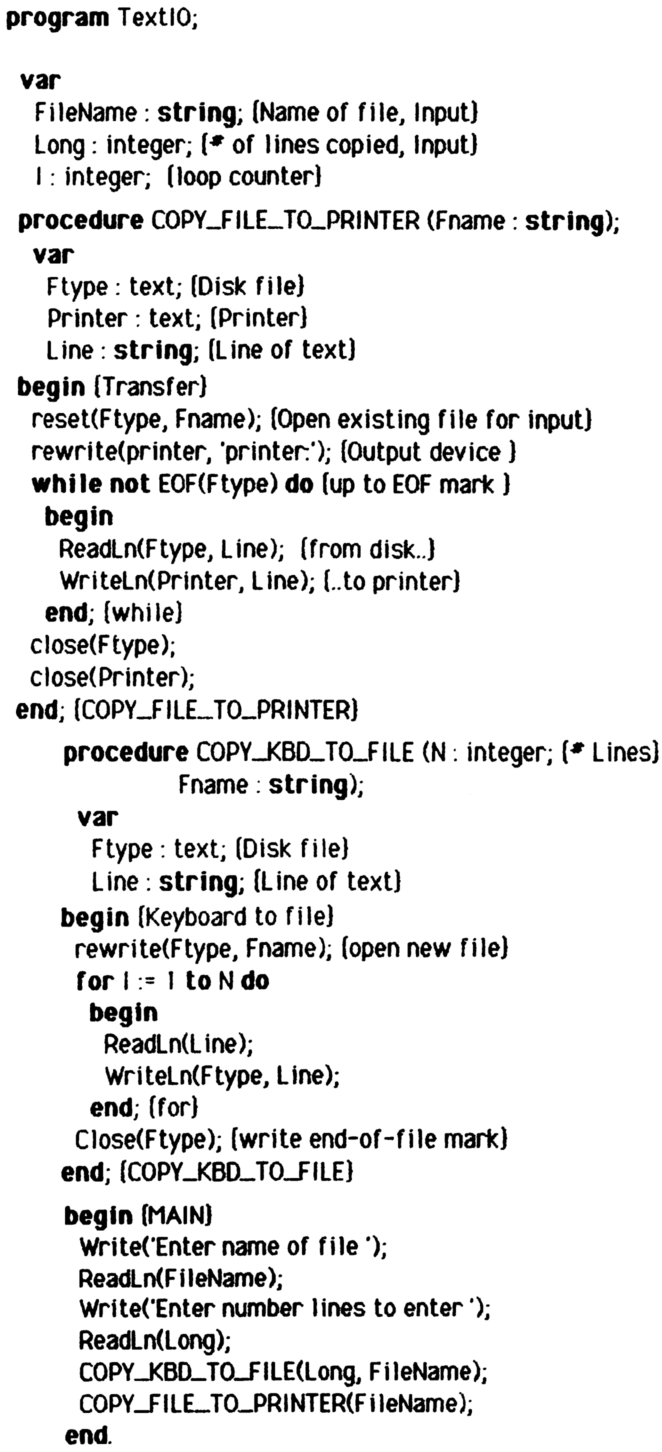
Notice that the printer is always opened with a Rewrite procedure. The Reset procedure is never used. Rewrite prepares a pseudo-file for output.

Hands-On Pseudo-File I/O

To demonstrate that text files and pseudo-files behave exactly the same, enter the program shown in Figure 9.3 and select RUN-GO.

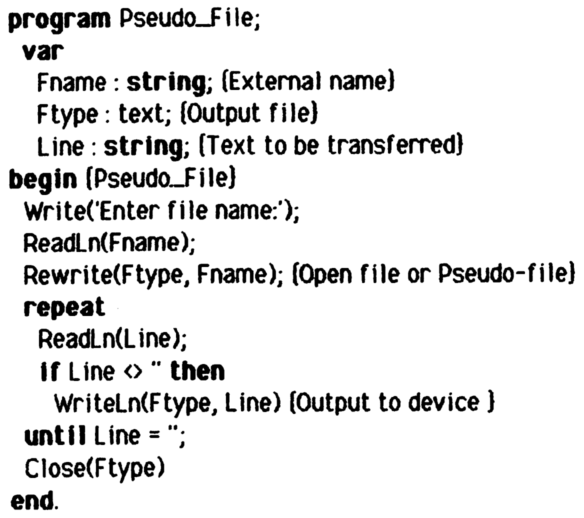
Enter DATA when asked to 'Enter filename' and type in several lines followed by a carriage return. To terminate the program, enter RETURN only (a null string).

Select RUN-GO again; this time enter PRINTER: for the name of the file. Now each line that you type is printed instead of copied to diskette. Terminate the program by entering a null string.



```
program TextIO;

 var
  FileName : string; {Name of file, Input}
  Long : integer; {# of lines copied, Input}
  I : integer;  {loop counter}
 procedure COPY_FILE_TO_PRINTER (Fname : string);
  var
   Ftype : text; {Disk file}
   Printer : text; {Printer}
   Line : string; {Line of text}
 begin {Transfer}
  reset(Ftype, Fname); {Open existing file for input}
  rewrite(printer, 'printer:'); {Output device }
  while not EOF(Ftype) do {up to EOF mark }
   begin
    ReadLn(Ftype, Line);  {from disk..}
    WriteLn(Printer, Line); {..to printer}
   end; {while}
  close(Ftype);
  close(Printer);
 end; {COPY_FILE_TO_PRINTER}
    procedure COPY_KBD_TO_FILE (N : integer; {# Lines}
            Fname : string);
     var
      Ftype : text; {Disk file}
      Line : string; {Line of text}
    begin {Keyboard to file}
     rewrite(Ftype, Fname); {open new file}
     for I := 1 to N do
      begin
       ReadLn(Line);
       WriteLn(Ftype, Line);
      end; {for}
     Close(Ftype); {write end-of-file mark}
    end; {COPY_KBD_TO_FILE}

    begin {MAIN}
     Write('Enter name of file ');
     ReadLn(FileName);
     Write('Enter number lines to enter ');
     ReadLn(Long);
     COPY_KBD_TO_FILE(Long, FileName);
     COPY_FILE_TO_PRINTER(FileName);
    end.
```

FIGURE 9.2
TextIO.



```
program Pseudo_File;
 var
  Fname : string; {External name}
  Ftype : text; {Output file}
  Line : string; {Text to be transferred}
begin {Pseudo_File}
 Write('Enter file name:');
 ReadLn(Fname);
 Rewrite(Ftype, Fname); {Open file or Pseudo-file}
 repeat
  ReadLn(Line);
  if Line <> '' then
   WriteLn(Ftype, Line) {Output to device }
 until Line = '';
 Close(Ftype)
end.
```

FIGURE 9.3
Pseudo-file.

This experiment shows how Macintosh Pascal handles devices as if they were files. Fname is the external name of either a diskette file or a device such as the printer or modem.

Study the If statement to understand how this program terminates.

If Line < > ' ' Then

As you can see, the single quotation marks have no blank space between them. Thus, Line < > ' ' will be True if you enter at least one character, and False if you press only RETURN. Similarly, the Repeat-Until terminates when Line = ' '.

Summary

There are two kinds of files in Macintosh Pascal: sequential and random access. In this session, you learned how to process a special kind of sequential file called a *text file*.

A text file is a collection of records; each record contains one line of text. The text lines may vary in length. An EOLN marker is used to signal the end-of-line. The final character of a text file is an EOF marker.

Devices behave like files in Macintosh Pascal. The printer is called PRINTER:, and the modem is called MODEM:. When devices act like text files, they are called *pseudo-files*. Anywhere a text file is used in a program, a device pseudo-file can be used as well.

All files must be declared in the Var statement, opened before being used, and closed after being used. The WriteLn statement is used to write a record; the ReadLn statement is used to input a record. The first identifier in the parameter list of WriteLn/ReadLn must be the internal name of the text file.

Problem Solving

1. Modify the COPY_KBD_TO_FILE procedure in Figure 9.2 to echo each line of input back to the Text Window in addition to writing the line to diskette.
2. Add a procedure called COPY_FILE_TO_SCREEN to the program of Figure 9.2 This procedure re-opens the diskette file and writes each record to the Text Window. Insert a call to this procedure immediately before the End statement in the main program.
3. What is the maximum number of records that a file can have?
4. What is EOLN? EOF?
5. How many EOF marks can a file contain?
6. What is the difference between a text file and any other sequential file?
7. Write a procedure called COPY_FILE_TO_FILE which copies the contents of an existing text file to a new text file. Your procedure will have two parameters: Fname1, Fname2.
8. Write a procedure called APPEND_TO_FILE which does the following:
   a. Opens an existing text file and reads to the end of the file (until EOF is sensed).
   b. Accepts lines of input text from the keyboard and appends (writes) them to the existing open diskette file.
   c. Quits when a null string is entered from the keyboard; closes the diskette file.
9. A text file cannot be shortened without writing a new file to replace it. Design and write a program to do the following:
   a. Open and read an existing file. Display each line read in the Text Window.
   b. Ask the user if each line is to be kept or deleted.
   c. If the line is to be deleted, skip over it and read the next line in sequence.
   d. If the line is to be kept, write the line to a new output file.
   e. When all records have been processed, close all files.
   f. Open all files and copy the new file over the old file.
   g. Close all files and quit.

Session 10:

Array Structures

In this session you will learn how to use objects called arrays *which can hold more than a single value. An array is declared in the Type or Var statement using the reserved keyword* array. *Arrays are the simplest form of a structured data type. You will learn the basis of structured types through examples of arrays.*

The Notion of an Array

In previous sessions you used objects to hold single values. For example, a character object holds a single character at a time; integer and real objects hold a single at a time; and so forth. An array, however, is a data structure for holding many objects at the same time.

An array is an object containing one or more values of the *same* type. Arrays are used to keep a list of items in memory under a single name. Each value in an array is called an array *element* or *item* and is associated with an array *index* or *designator.* The index is used to reference the individual items in the array and can be any expression which evaluates to an ordinal value (char, integer, Boolean).

The length of an array is fixed by the programmer when the array identifier is declared in a Var statement. The beginning and ending values of the index are explicitly given, as shown by the examples below.

```
Var
   LIST : Array [1 .. 10] of Integer;
```

This declares LIST as an array of 10 integers; each integer is designated by an index 1 .. 10. Another way to declare LIST is as follows.

```
Type
   LISTS = Array [1 .. 10] of Integer;
Var
   LIST : LISTS;
```

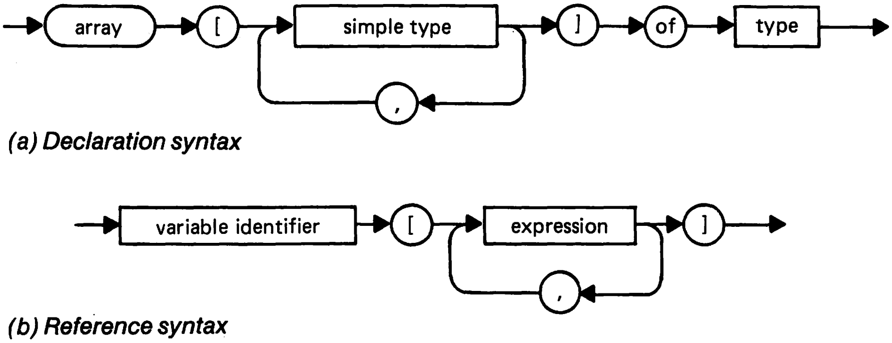
The length of an array can be set in a Const statement. Here are two ways of doing this:

```
Const
   N = 10;
Var
   LIST : Array [1 .. N] of integer;
```

A second method, which accomplishes the same thing, is perhaps more elegant.

```
Const
   N = 10;
Type
   INDEXES = 1 .. N;
   LISTS = Array [INDEXES] of Integer;
Var
   LISTS : LISTS;
```

The syntax diagram of an array type declaration is shown in Figure 10.1 (a). Square brackets are used to enclose the index part of an array. Figure 10.1 (b) shows the syntax of a reference to an array, for example, if the array is used in an expression.



(a) Declaration syntax

(b) Reference syntax

FIGURE 10.1
Array syntax diagrams

Examples of Arrays

An array is a collection of values all of the same type; each value is designated by an index. The index need only evaluate to an ordinal. Integer and char indexes are both permitted. Here are some examples.

```
Type
   INDX = 'a' .. 'z';
   INDY = 0 .. 9;
Var
   LETTERS : Array [INDX] of Char;
   NUMS    : Array [INDY] of Integer;
```

These can be used to hold and index values as illustrated by the assignment statements below.

```
LETTERS ['a'] := 'A';
LETTERS ['b'] := 'B';
LETTERS ['y'] := 'Y';
```

Also,

```
   NUMS [0] := 0;
   NUMS [1] := 1;
   NUMS [9] := 9;
```

Tables

Suppose you want to convert the vowels from character to integer values. A conversion table might be used as follows:

Vowel	*Integer*
a	1
e	2
i	3
o	4
u	5

The Pascal equivalent of this table might look as follows:

```
Type
    VOWELS = Array ['a' .. 'u'] of 1 .. 5;
Var
    VOWEL : VOWELS;
```

The table must initially be set up in the body of the program by assigning each of the array elements a number.

```
VOWEL ['a'] := 1;
VOWEL ['e'] := 2;
VOWEL ['i']  := 3;
VOWEL ['o'] := 4;
VOWEL ['u'] := 5;
```

A table similar to this one is used in the next hands-on example. Notice how subrange 1 .. 5 is used to declare the type of each element of the array. Only operations compatible with the base type are permitted on the array.

The array structure is commonly used to store tables in memory. In the next hands-on experiment you will use an array to store a table.

Hands-On Arrays

Suppose you want a function called "BELONGS (Ch : Char; Var V : TABLE): Boolean" which is True if "Ch" is a vowel and False otherwise. The body of BELONGS compares Ch with each of the characters stored in an array of vowels called "V". If a match occurs, the value True is returned (see Figure 10.2).

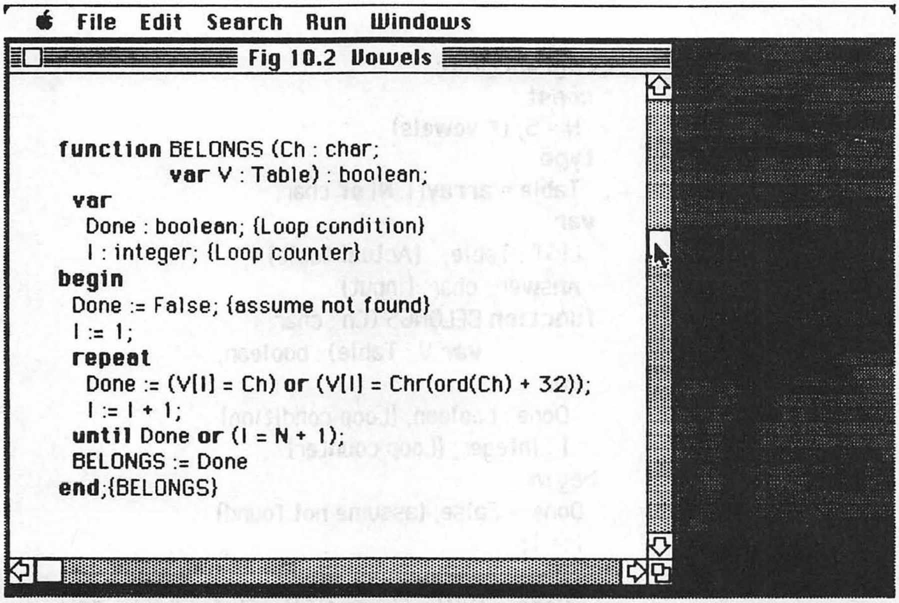
A second subprogram, shown in Figure 10.3, is needed to initialize the table of vowels. Procedure INIT_TABLE simply assigns values to each element of the array and returns these to the main program.

Enter both of these subprograms into the Program Window along with the program heading and data declaration parts:

```
Program MAIN;
  Const
     N = 5;
  Type
     TABLE = Array [1 .. N] of Char;
  Var
     LIST    : TABLE;
     Answer : Char;      {input}
```

Now add the body of the main program as follows, and select RUN-GO.

```
Begin       {MAIN}
   INIT_TABLE (LIST);
   WriteLn ('Enter a letter: ');
   ReadLn (Answer);
   If BELONGS (Answer, LIST) Then
            WriteLn ('It's a vowel!')
   Else
            WriteLn ('It's a boy!')
End.        {MAIN}
```



```
function BELONGS (Ch : char;
          var V : Table) : boolean;
 var
  Done : boolean; {Loop condition}
  I : integer; {Loop counter}
begin
 Done := False; {assume not found}
 I := 1;
 repeat
  Done := (V[I] = Ch) or (V[I] = Chr(ord(Ch) + 32));
  I := I + 1;
 until Done or (I = N + 1);
 BELONGS := Done
end;{BELONGS}
```

FIGURE 10.2
Function BELONGS.

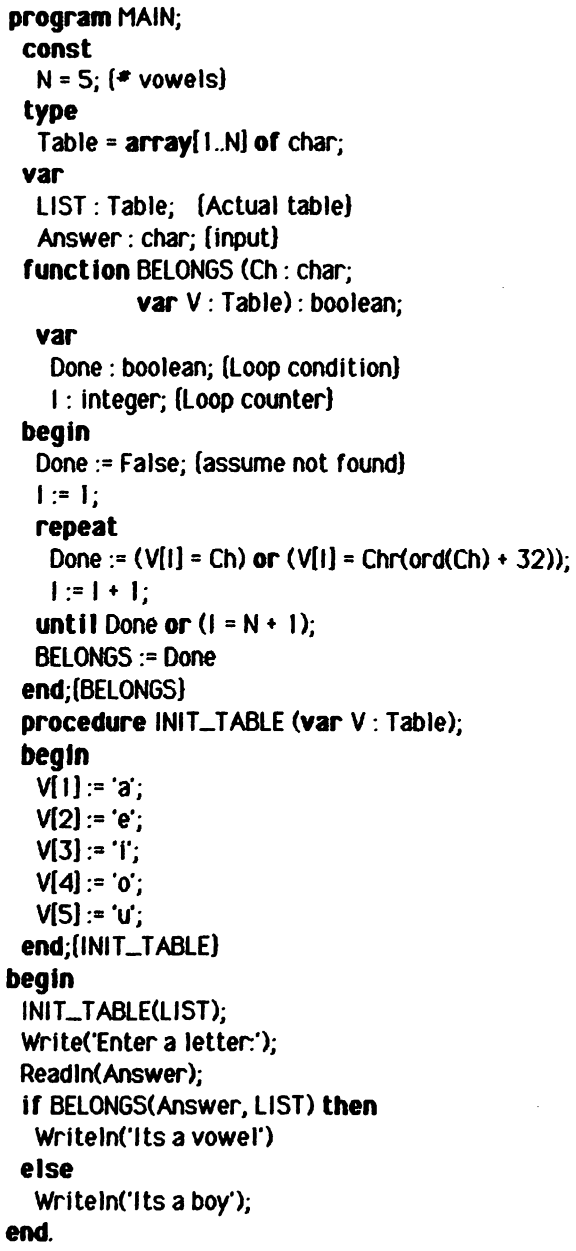
Open the Text Window so you can see the results of running this program. Enter a capital letter A and you should get “It's a vowel!”. Enter a 'z' and you should see “It's a boy!” appear in the Text Window.

Lists

One of the most common uses of arrays is for keeping lists. In this section you will learn how to enter, sort, and display a list of phone numbers. The sort technique used here is well known as a simple, but inefficient method. For our purposes, this method will be good enough, but be warned that it should not be used on very large lists.

Suppose you want to keep a small list of telephone numbers in a table or electronic phone book. Each phone book entry consists of an extension phone number.

1212
5515
8150
5080



```
program MAIN;
 const
  N = 5; {# vowels}
 type
  Table = array[1..N] of char;
 var
  LIST : Table;   {Actual table}
  Answer : char; {input}
 function BELONGS (Ch : char;
         var V : Table) : boolean;
  var
   Done : boolean; {Loop condition}
   I : integer; {Loop counter}
 begin
  Done := False; {assume not found}
  I := 1;
  repeat
   Done := (V[I] = Ch) or (V[I] = Chr(ord(Ch) + 32));
   I := I + 1;
  until Done or (I = N + 1);
  BELONGS := Done
 end;{BELONGS}
 procedure INIT_TABLE (var V : Table);
 begin
  V[1] := 'a';
  V[2] := 'e';
  V[3] := 'i';
  V[4] := 'o';
  V[5] := 'u';
 end;{INIT_TABLE}
begin
 INIT_TABLE(LIST);
 Write('Enter a letter:');
 Readln(Answer);
 if BELONGS(Answer, LIST) then
  Writeln('Its a vowel')
 else
  Writeln('Its a boy');
end.
```

FIGURE 10.3
Complete program to search for vowels.

Obviously this simple example is not very useful because we have dropped off the person's name, but it will serve as a basis which can be expanded on in later sessions.

The maximum length of the list must be declared first, and then the list itself declared as an array. The following is one of many possible methods of declaring a list.

```
Const
   Max_Size = 10;
Type
   Entries = integer;        {can be any type you want}
   Book    = Array (1 .. Max_Size] of Entries;
Var
   PHBook : Book;
```

PHBook is an array of integers; each integer contains an extension number. Since the elements of PHBook are integers, the base type of PHBook is integer.

The phone book program shown in Figure 10.4 contains three procedures for operating on the PHBook list. Procedure Add is called to append an additional entry to the end of the existing list; Procedure Lookup searches the PHBook, looking for a matching entry, and Procedure Sort orders the entries into alphabetical order. (Note that Lookup has been left as an exercise for you to complete.)

Each of these procedures is called from the case clause shown in the main program body. The main body simply prompts for your request, then calls the appropriate procedure to do the work.

Add to the List

Procedure Add reads an integer such as the following:

```
1212
```

and appends it to the end of PHBook. The index of the last entry in PHBook is LAST. If incrementing LAST by one would cause it to exceed the length of the list, an error message is displayed: "Sorry, no more room". If there is enough room, LAST is increased by one and the new entry is stored in PHBook [LAST].

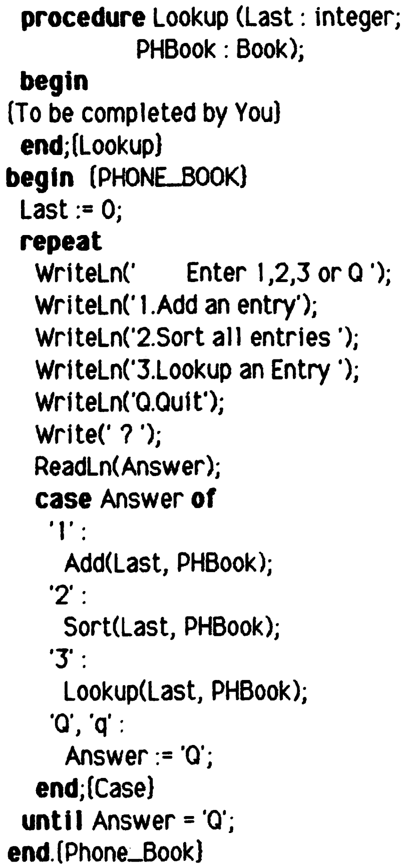
Initially, LAST is zero (see the main program body). Each new entry bumps LAST up by one, until it eventually reaches Max_Size. Keep in mind that the list must be entered each time the program is run. Saving the program does not save the values stored in PHBook.

Sort the List

Procedure Sort rearranges the list of integers so that they are in increasing order. The algorithm used by Sort is given in words

```
program Phone_Book;
 const
  Max_Size = 20; {Up to 20 entries}
 type
  Entries = integer;{whole numbers, only}
  Book = array[1..Max_Size] of Entries;
 var
  PHBook : Book;
  Last : integer; {index of last entry}
  Answer : Char; {Input}
 procedure Sort (Last : integer;
          var PHBook : Book);
  var
   I, J, Small : 1..Max_Size;{working counters}
   Temp : Entries;{temporary place}
 begin
  for I := 1 to (Last - 1) do { order sub-list, I..Last-1}
   begin
    Small := I;
    for J := I + 1 to Last do
     if PHBook[J] < PHBook[Small] then
      Small := J; {current smallest entry}
    Temp := PHBook[Small];
    PHBook[Small] := PHBook[I];
    PHBook[I] := Temp; {exchange smallest with top o'list}
   end;{For I}
 end;{Sort}
 procedure Add (var Last : integer;
          var PHBook : Book);
  var
   Entry : Entries; {Phone Number}
 begin
  Write('Enter entry:');
  ReadLn(Entry);
  if Last < Max_Size then
   begin
    Last := succ(Last); {grow another entry}
    PHBOOK[Last] := Entry
   end{if-then}
  else
   WriteLn('Sorry, no more room');
 end;{Add}
```

FIGURE 10.4
Program Phone_Book.



```
  procedure Lookup (Last : integer;
                PHBook : Book);
  begin
 {To be completed by You}
  end;{Lookup}
 begin  {PHONE_BOOK}
  Last := 0;
  repeat
   WriteLn('      Enter 1,2,3 or Q ');
   WriteLn('1.Add an entry');
   WriteLn('2.Sort all entries ');
   WriteLn('3.Lookup an Entry ');
   WriteLn('Q.Quit');
   Write(' ? ');
   ReadLn(Answer);
   case Answer of
    '1' :
     Add(Last, PHBook);
    '2' :
     Sort(Last, PHBook);
    '3' :
     Lookup(Last, PHBook);
    'Q', 'q' :
     Answer := 'Q';
   end;{Case}
  until Answer = 'Q';
 end.{Phone_Book}
```

***FIGURE 10.4** (continued)*

1. Starting with "I := 1" and "For I:=1 to LAST-1", do the following:
   a. Assume Small is the index of the smallest integer in the list.
   b. Search the remaining elements in " I + 1 to LAST " looking for an element smaller than element Small.
   c. If a smaller than element Small is found, make it the smallest element.
   d. Exchange the smallest element found with the first element "I" in the list.
2. Each time "I" is incremented by one, the sublist to be ordered becomes shorter. Finally, only one integer remains, so you are done.

After you have entered three or four integers into the list, select the Sort operation and study it. Pull down WINDOWS-OBSERVE and watch LAST, I, J, SMALL, PHBook[1], PHBook[2], and PHBook[3] change values as the sorting takes place. Use RUN-STEP-STEP to step through the example.

Procedure Sort may be difficult to follow due to its nested For and If statements. Carefully study the indented statements

```
For J := I + 1 To LAST Do
    If PHBook [J]
        Small : = . . . . . . . . .
```

These are nested, one within the other. The assignment statement is the only statement in the If-Then clause, and there is *no* Else clause. The If-Then statement is the only statement in the body of the For loop.

These three nested statements find the index of the smallest integer within the remaining sublist. The next three statements exchange the smallest integer with the first integer in the sublist.

Look Up an Element in a List

The Lookup procedure has been left as an exercise. Its purpose is to search PHBook looking for an Entry. If found, the entry is displayed, but how can you search for something if you don't already know what it is?

A *key* is a portion of a list element that uniquely identifies the element. We need a unique key to help search for phone numbers in PHBook. A person's name is probably the best candidate for a key, but the simple phone book example does not store names. Just for simplicity, assume that the extension number is the key.

A general procedure or function for searching an array would look like the following:

```
I:=0;                          {loop counter}
Found:=False;                  {loop condition}
while ( not Found )
          and
      ( I <= Last )            {Last is the length of the array}
  do begin
      "
      "
      "
  I:=I + 1;                    {increment loop counter}
end;
```

Remember, the loop can terminate either because the number has been found or else because the entire list has been searched without finding the desired number.

Two-Dimensional Arrays

A two-dimensional array is an array with two indexes. The two values are separated by a comma, as shown below.

```
Type
  ONE = 1 .. 5:
  TWO = 1 .. 10;
Var
  Two_D : Array [ONE, TWO] of Integer;
```

Both index values must be specified whenever referencing one of the elements.

```
TWO_D [I, J] := TWO_D [5, 9] + TWO_D [1, 3];
```

Arrays are not limited to only one or two dimensions, but may be multidimensional.

Summary

The Sort and Lookup procedures of this session have illustrated several of the more common operations on arrays. In general, arrays are used to keep lists in memory while they are searched, sorted, and manipulated through procedures that insert, delete, and change the elements of the list.

You will almost always use a loop to process arrays. The loop counter is typically used as the array index. Be careful not to "run off the end" of an array when reading, writing, or searching it. The loop counter should be the same type as the array index in order to prevent exceeding the array index bounds, as shown below.

```
const
  Max = 100;
type
  sub = 1 .. Max;
var
  i: sub;
  A : array [ sub ] of blob;        {i and sub same range}
```

Any ordinal type can be used as an array index. Hence, an index can be a character, Boolean, or integer. This feature is convenient and helpful when an array is used to store a table.

An array is the simplest form of a structured type in Pascal. It is also the simplest way to store a large number of elements under a single name. But your program must know the length of an array prior to running. This restriction can be removed by using dynamic storage, as discussed in Session 16.

Problem Solving

1. Write the Lookup procedure discussed in this session and add it to Program Phone_Book in Figure 10.4. Try it.
2. Write a program that reads a list of N numbers into memory and then computes their average value. Use the following, where appropriate:

```
var
   Max = 50;                         {maximum value of N}
   For I := 1 To N Do
       ReadLn      (LIST [I]);
   For I := 1 To N Do
       WriteLn (LIST [I]);
```

3. Add a DISPLAY procedure to the hands-on example of Figure 10.4 that writes all elements of PHBook to the Text Window.
4. Add a DELETE procedure which removes an entry from PHBook of Figure 10.4. The vacated element must be closed in by moving all elements beyond the deleted element. Your procedure should contain statements to move elements from the J-th position up.

```
For K := J To (LAST-1) Do
    PHBook [J] := PHBook [J + 1];
LAST := Pred (LAST);
```

5. Write loops to do the following:
   a. Fill an array of characters with blanks.
   b. Fill an array of integers with zeros.
   c. Multiply each element of an array of integers by 2.
   d. Copy the even numbered elements from one array to the consecutively numbered elements of another array.
   e. Reverse the elements of an array.
   f. Find the largest element of an array of integers.
   g. Count how many elements of an array of integers are greater than integer B.

Session 11:

String Processing

In this session you will learn how to write programs that process text rather than numbers. A string processing program stores, reads, writes, and uses intrinsic string processing functions and procedures to insert, delete, search, and move strings of text within computer memory.

Strings in Pascal

The seventh-century story of *Beowulf* is the oldest known story in English literature. It became a written document during the time of King Alfred (840-899). However, Geoffrey Chaucer (1340-1400) was primarily responsible for establishing English as a written language, even though *The Canterbury Tales* had to be copied by hand.

Manipulation of text is one of the most important applications of personal computers, word processing being the most obvious example. All text processing programs work with a special kind of array structure called a *string*. A string is an array of characters; it has an attribute called *length* and a set of allowable operations that work with it. This chapter will show you how to represent and process strings from within a Pascal program.

In Pascal, a sequence of zero or more characters between two single quotation marks is called a *string*. You have already used strings as prompts in Write and WriteLn statements.

```
WriteLn ('This is a character string.');
```

The statement above is an example of a *literal string*, but *string constants* and *variables* are also permitted in Macintosh Pascal. (Standard Pascal does not have a data type called String.)

String Declaration

Defining variables of type String is like defining variables of any other type. The following Const and Var statements define one constant and two variables of type String:

```
Const
     Ans = 'Yes';                 {constant string}
Var
     String1 : String;            {up to 255 chars}
     Str2    : String[20]         {up to 20 chars}
```

A constant whose value is two or more characters constitutes a string. If ANS ='Y' it is a char constant, because it contains a single character. Therefore Const ANS ='Yes' declares ANS as a string, rather than a char object.

Variables String1 and Str2 illustrate two different methods of defining variables of type String. String1 is a variable which can hold strings whose length can vary up to the maximum default value of 255 characters. Str2 is defined as a string valued variable which can hold up to 20 characters. The constant enclosed in square brackets specifies an upper limit to the length of a string. Unless the maximum length is specified, Pascal will assume 255 characters.

Length of Strings

Each variable of type String can hold a varying number of characters. For example, after execution of the following assignment statement, the length of str2 is five.

```
str2     :=     'Minoo';          {assign 5 characters}
```

Obviously, if we delete all the characters from a string variable, its length becomes zero. A string is called *null* or *empty* if its length is zero.

```
str2     := '';          {no blanks or characters}
```

Notice that an empty string is different from a string containing blank characters. When a Pascal program begins to run, the length and contents of all string variables are undefined. Do not assume merely because you have declared a variable of type String that its length stays at zero, since you have not yet assigned any value to it. If you want to know the length of a string variable, you must use the *length* function to find it out.

The Difference Between String Size and String Length

It is very important to distinguish between the size and length of a string. The actual number of characters in the string variable at any time during the execution of a program is called its *length*. The *size* of a string-type variable remains static throughout the execution of your program and is defined when you declare the string variable. If at the time of declaration no size is specified, then the size of the string variable is assumed to be 255. The size of str2 is 20, which means the length of the string value in str2 cannot exceed 20. As a rule, length may vary from 0 to the size, whereas size is always fixed by the Const or Var declaration.

Finding the Length of a String

To find the exact number of characters that a string variable holds, use the intrinsic Length function.

```
WriteLn (Length (str2) ) ;
```

If str2 contains 'Minoo', the WriteLn will cause five to be printed in the Text Window. The length function returns an integer value equal to the current length of its string parameter. Length may vary from zero (null string) to 255 (maximum length).

Hands-On String Length

Start your Macintosh Pascal and enter the following declarations to define str and str1 as String variables.

```
Var
      str, str1 : String;
```

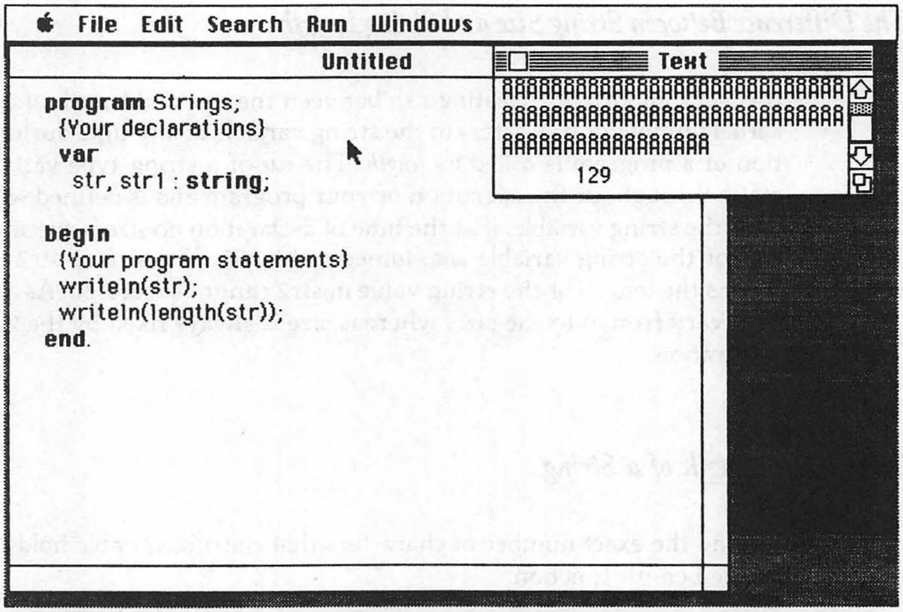
Now add the following to the main body of the program and select RUN-GO.

```
WriteLn (str);                  {find initial value of str}
WriteLn ( Length (str1) ) ;     {find initial length of str1}
```

The Text Window of Figure 11.1 shows the result of running this program. You may get something different, depending on what happens to be in main memory at the time this example is run. The result you see confirms our warning that you cannot assume the length of a string variable to be zero or its contents to be null (empty) simply because it has been declared.

Insert the following statement above the first WriteLn statement, then select RUN-GO.

```
str := '';           {two single quotes}
```



```
program Strings;
{Your declarations}
var
  str, str1 : string;

begin
{Your program statements}
writeln(str);
writeln(length(str));
end.
```

```
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAAA
AAAAAAAAAAAAAAAAAA
          129
```

FIGURE 11.1
Contents and length of str before initialization.

This time, you get zero as the length of str and nothing as its value. You *initialized* str to empty and at the same time set its length to zero. This illustrates the following important rule:

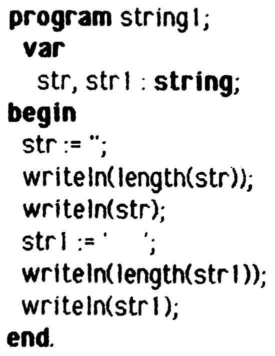
String Assignment:	Whenever a value is assigned to a string variable, the length of the string is automatically changed to equal the number of characters assigned to the string.

Now add the following statements after the last WriteLn statement and RUN-GO again:

```
str1 := '     ';                {5 blank spaces}
WriteLn ( Length(str1) );
WriteLn (str1) ;
```

This time, you get five for the length of str1 and five blanks as its contents, even though the blanks cannot be seen in the Text Window.

This example shows the subtle differences between an empty or null string variable and a string variable which contains blank characters. Also, it should be evident by now that the Length intrinsic function returns an integer value for the length of a string variable. Figure 11.2 lists the resulting program.



```
program string1;
 var
   str, str1 : string;
begin
 str := '';
 writeln(length(str));
 writeln(str);
 str1 := '   ';
 writeln(length(str1));
 writeln(str1);
end.
```

FIGURE 11.2
Program listing to display the difference between empty string variable and string variable containing blanks.

String Operations

Macintosh Pascal provides many intrinsic functions that can be used to manipulate the contents of string variables.

Note:
S, T, PATTERN: are strings
Index, Count: are integers

Function	*Explanation of Function*
LENGTH (S)	Returns the current length (integer value) of string S.
POS (Pattern, S)	Returns zero if PATTERN cannot be found as a substring within S. Returns the integer location (index) of the first character of the first occurrence of PATTERN within S. The location of characters in S are numbers from 1 to Length (S).
CONCAT (S, T)	Returns a string which is the value of T concatenated (appended) to the tail of S. The resulting string is of length Length (S) + Length (T). CONCAT may have more than two arguments: CONCAT(S,T,U).
COPY (S, Index, Count)	Returns a string containing the Count characters of S, beginning with character Index. If count < = 0, returns null string. If (Count + Index) > Length (S), returns the tail of S.
DELETE (S, Index, Count)	This is a procedure instead of a function, which modifies S by removing Count characters from S beginning with

character S[Index]. If fewer than Count characters are in the string beginning with position Index, then the remaining characters are removed.

OMIT (S, Index, Count) The same as DELETE except OMIT is a function which returns a string containing the result of deletion of Count characters from S. The value of S is *not* changed.

INSERT (S, T, Index) This is a procedure instead of a function which modifies T. String S is inserted into string T at character position Index. The first character of S is put into T[Index], the second into T[Index + 1], and so forth. If Index < 1 the result is as if S were appended to T by concatenation: T := CONCAT (S,T). If Index > Length (T) the result is T := CONCAT (T,S).

INCLUDE (S, T, Index) This is the same as INSERT except INCLUDE is a function which returns the result as a string. The result is a string which contains T with S inserted as described in INSERT. S and T are not modified.

Searching Strings

Suppose you want to write a program to find the number of words in a sentence. To do this, your program must first locate each word in the sentence and then count them. Such a program would print 6 for the number of words in the sentence stored in the following string variable.

```
str := 'This is a six word sentence.';
```

One way to count the number of words in a sentence is to search for and count the number of blank spaces between words in the sentence. All but the last word are followed by blank characters, so the number of words is one greater than the number of blank spaces in the sentence. The following algorithm counts words using this premise. (Notice that a space must consist of only one blank character for this premise to hold up.)

```
Set No. of words to zero
Look for words in the sentence:
    If a word is found
        Then repeat until no more blanks:
                  add 1 to the number of words
                  look for the next blank
        Otherwise add 1 to the number of words
Print the number of words found.
```

This algorithm can be implemented in Macintosh Pascal using intrinsic string functions, which are discussed next.

Looking for Patterns

The POS function returns the position of the *first* occurrence of a substring (in our example, blank characters) in a string. To see how this works, start Macintosh Pascal and enter the following in the Program Window:

```
Var
            str, str1  : String;
            index : Integer;
Begin
            str := 'This is a six word sentence';
            WriteLn  ( Length(str) ) ;
            index        := POS  (' ',str);
            WriteLn  (index)
End.
```

Now select RUN-GO. As you can see, this program prints 27 for the length of str and 5 as the position of the first occurrence of a blank. In general, you can use POS to find the starting position of any pattern in a string.

As another example, let's see what happens if you try to find the position of "is" in the current value of str. Replace:

```
index := POS  ('       ', str);
```

with:

```
index := POS  ('is',str);
```

Now RUN-GO. You may be surprised to get 3 instead of 6, although the word "is" starts at position six. This example should remind you that POS starts searching from the beginning of the string; when it finds the first instance of "is" in "This," the value of 3 is returned. If you want to find the word "is", you must pass POS a unique pattern so that it continues to search for the desired substring. To see how to find the position of the word "is" in the sentence, change the POS statement as follows:

```
index := POS  (' is',str) + 1;        {blank}
```

Now RUN this new version. Voila! You get it right this time. The result is 6.

Actually, in order to get the right result, we cheated a little bit. The pattern we looked for starts at postion five because of the character blank at its beginning. We continually added 1 to help POS return the correct position of "is".

Deleting Substrings

Recall the program for counting the number of words in a sentence. POS by itself does not work. No matter how many times we use POS, it always returns the position of only the first blank and does not continue to locate the others.

If each word is removed after it is found, then subsequent application of POS finds the next word. Repeating, each new word is located, removed, and counted until all words have been counted. For example, after the first word is removed from position one the string looks like the following:

```
'is a six word sentence'
```

If you apply POS to the new string, you get 3 as the position of the next blank character. This blank is the second blank character in the original sentence.

Use the *Delete* procedure to delete a portion of a string. The general format of Delete is as follows:

```
Delete (dest,index,Count);
```

dest is the name of the string variable that you want to delete characters from (dest = destination).

index is an expression of type Integer which specifies the starting position of what you want to be deleted.

Count is an expression of type Integer that specifies the number of characters to be deleted.

For example,

```
Delete (str,1,10);
```

deletes the first ten characters of the value of string variable str. If we assume that the current value of str is:

```
'This is a six word sentence'
```

what remains in str after the Delete operation is

```
'six word sentence'
```

To delete each word as it is found, insert the following after the POS statement in Program WordCnt1:

```
Delete (str,1,index);
```

This causes all characters in str to be deleted starting from the beginning of str, counting up to index. The first time the program executes Delete, index is 5 and 'This ' is removed from str. The contents of str after this deletion are shown below.

```
'is a six word sentence'
```

A careful application of POS and Delete, repeatedly, can help you count the number of words in any sentence with one minor exception.

More on POS

What if there is no blank character in a sentence? Perhaps the sentence contains only one word or the string variable is null. In cases like this, POS cannot find a matching pattern, so POS returns zero. A zero means no pattern was found in the specified string.

Your word counting program will continue to find the position of blanks and delete the corresponding words. Eventually, you get to the point where the only value left in str is 'sentence'. If POS is applied one more time to str it will return zero. To see this clearly, make the following additions to Program WordCnt1 and RUN-GO.

```
str       := 'sentence'                   {in case substring}
WriteLn  ( POS('      ',str1) ;           {blank doesn't exist}

str       := '';                          {in case there is nothing}
WriteLn  ( POS('      ',str1) ;           {in the string variable}
```

POS returns zero in both cases. Figure 11.3 shows the first version of a program that counts the number of words in str.

An Improvement to the Hands-On Program

Let's expand Program WordCnt1 (Figure 11.3) to print the sentence as well as the number of words in it. As an example, the output should look like the following:

```
This is a six word sentence          6
```

As a first attempt to modify the program, suppose you change the WriteLn statement which prints the number of words to the following:

```
WriteLn  (str,NoOfWords) ;
```

When the modified program is run, you get the following incorrect output:

```
sentence          6
```

The program repeatedly removed words until the value of str shrank to the last word in the original sentence.

There are two solutions to this problem. The better solution is just to make a copy of the whole sentence, keep it separate, and at the end print it along with the number of words. To do so, you need to add

```
str1  := str;             {to make a duplicate of str}
```

after

```
str  := 'This is a six word sentence';
```

```
program WordCnt1;
 {Your declarations}
 var
  str, str1 : string;
  index : integer;
  NoOfWords : integer;
begin
 {Your program statements}
 str := 'This is a six word sentence';
 if length(str) = 0 then { for the case that str is empty }
  NoOfWords := 0
 else
  begin
   index := Pos(' ', str);
   while (index > 0) do
    begin
     NoOfWords := NoOfWords + 1;
     Delete(str, 1, index);
     index := Pos(' ', str);
    end;
   NoOfWords := NoOfWords + 1;
  end;
 writeln('No of words in str is: ', NoOfWords);
end.
```

FIGURE 11.3
Program listing for first version of Word Count program.

and change the last WriteLn statement to:

```
WriteLn (str1,NoOfWords);
```

Making these changes solves the problem, but for the sake of argument, suppose you don't know about this simple solution. An alternate way of achieving the same effect is to copy the deleted words into another string as the deletions take place. Then, at the end, copy the last word left in str to the tail of the new string variable.

Copy and Concat

The Copy and Concat functions can be used to copy the word to be deleted from the original string into a temporary string variable and then append the saved word to another string variable. When the program has removed all words from the original string, the other string variable will contain the original sentence.

```
program WordCnt2;
  {Your declarations}
  var
    str, str1, str2 : string;
    index : integer;
    NoOfWords : integer;
begin
  {Your program statements}
  str := 'This is a six word sentence';
  if length(str) = 0 then { for the case that str is empty }
    NoOfWords := 0
  else
    begin
      index := Pos(' ', str);
      while (index > 0) do
        begin
          NoOfWords := NoOfWords + 1;
          str2 := Copy(str, 1, index);
          str1 := Concat(str1, str2);
          Delete(str, 1, index);
          index := Pos(' ', str);
        end;
      NoOfWords := NoOfWords + 1;
      str1 := Concat(str1, str);
    end;
  writeln(str1, NoOfWords);
end.
```

FIGURE 11.4
Program listing for second version of Word Count program.

The following changes should be inserted before the Delete statement in Program WordCnt1:

```
str2 := Copy (str,1,index);       {extract a word}
str1 := Concat (str1,str2);       {append to tail of str1}
```

Also, after the end of the While statement, add:

```
str1 := Concat (str1,str);
```

After these additions have been made, run the program. Figure 11.4 shows this second version of the word counting program, called Program WordCnt2.

We can now look at the details of Copying and Concatenating strings. The general format of the Copy function is:

```
Copy (str,index,Count);
```

str is a string variable or string value.
index is an Integer valued expression designating the starting location for the Copy operator.
Count is an Integer valued expression designating the number of characters to be copied.

Here are some examples using the Copy operator:

```
str := 'This is an example of Copy operators';
Copy (str,9,10);          {returns 'an example'}
Copy (str,20,50);         {returns 'of Copy operator' }
Copy (str.0,6);           {returns 'This is'}
```

Notice in the second example, the Count value (50) is beyond the last character of the current value of str. When the Count is larger than what the string variable holds, Copy only returns as many characters as are available. In the third example, the starting index was zero. If you specify 0 at the starting location, Copy assumes you mean to start with the first character of str.

Concat pastes two or more strings together. For example, assume the following:

```
str1 := 'Hi';
str2 := 'there ' ;
str3 := 'you ';
str4 := ";
```

str4 before		*str4 after*	*Length (str^4)*
(empty)	str4 := Concat (str1, str2)	Hi there	8
Hi there	str4 := Concat (str4, str3)	Hi there you	12
Hi there you	str4 := Concat (str1, str2, str3)	Hi there you	12
Hi there you	str4 := Concat (str4, str1, str2)	Hi there you hi there	20

Inserting Strings

Concat is useful for appending one string to the tail of another, but what if you want to add one string to the middle of another string? For example, suppose

```
str1 := 'This an example of insertion'
```

To add "is" to the sentence, you can use the *Insert* intrinsic procedure. The general format of Insert is:

```
Insert (source, dest, index);
```

source is the character string you want to insert into dest.
dest is the character string you want to insert source into.
index is the starting position for insertion in dest.

For example, to insert 'is' into character position 6 of str1, use the following procedure call:

```
Insert ('is ',str1,6);
```

Insert puts the contents of the first string variable (the source) into the second string variable (the destination), starting at the location specified by the index. Whatever is already in position index of str1 is shifted to the right to make room for the inserted characters. Following are more examples showing the value of strings before and after the Insert operation.

```
                    ( before --> after).

          Insert (' John', str1, Length(str1) + 1);
'Here is my friend'-->'Here is my friend John'

          Insert ('old ', str, 12);
'Here is my friend John'--->'Here is my old friend John'

          Insert ('May ',str, 1);
'I introduce my old friend John?'--->'May I introduce my old friend John?'
```

Summary

This session covered the principal functions and procedures used to manipulate character string values. Strings have a length, which is the number of characters stores in a string. There is a difference between empty or null strings and strings containing blanks. A null string has a length of zero. A string containing blanks actually contains characters. There are two ways to declare string variables: (1) by specifying the maximum length, for example, [20], and (2) by letting the system use the default size of 255.

Strings are input and output just like any other value. For example, ReadLn (STR1) is used to read a string from the keyboard and WriteLn (STR1) writes the value of STR1 to the Text Window.

Problem Solving

1. Write a new delete procedure using other intrinsic functions and procedures. (*Hint:* Use the Copy and Concat functions.)

2. Change the Hands-On example to use your own Delete procedure instead of the Macintosh Pascal Delete procedures.
3. Write a program that counts the number of nonblank characters in a sentence and displays the count next to the original sentence. Use string functions to do this problem.
4. Modify the Hands-On example to read its input from the Text Window (ReadLn (STR);) instead of assigning the value in the program.
5. Modify the program in Problem 4 to continue reading sentences from the keyboard and printing the number of words found in each sentence until the length of the input string read is zero.

Session 12:

Data Structures Containing Records

In this session you will learn how to create new data types containing nested components of (possibly) different types called records. *You will also learn how to use Pascal's With statement to increase program efficiency and reduce programming effort.*

The Structure of Data

In the early 1940s, Professor J. W. Mauchly believed that weather prediction was possible if only a machine could be developed to rapidly solve mathematical problems involving thousands of numbers. He and another University of Pennsylvania professor, J. Presper Eckert, invented and constructed the first full-scale working electronic digital computer to do 100 years of calculations in 2 hours. The ENIAC (Electronic Numerical Integrator and Computer) had 18,000 vacuum tubes and consumed as much power as 15,000 Macintosh computers, but it could perform 360 multiplications per second!

The early electronic computers were constructed for the sole purpose of doing calculations on numbers, but today's problems often require nonnumerical processing. Text, pictures, and sound are also processed by computers. The word "data" no longer means long lists of numbers, but includes any representation of information.

A *data structure* is any logical structure for organizing information so that it can be processed by a machine. Macintosh Pascal has several mechanisms for constructing data structures. You have already used the simplest form of data structure: constant and variable objects which can be identified by name. Now you will be introduced to structures for holding data of differing types.

The Record Structure

In the previous session you learned that an array was a collection of values of the same type. In this session you will learn about a structure that is a collection of values of possibly different types. A *record* is an object with components. A component may be either another record structure or an object of a base type. In a sense, a record is a nested data structure.

A record object is declared in Pascal by listing its components and their types between a Record-End pair of keywords.

```
Var
    PERSON : Record                                  {start record}
                   AGE    : Integer;                 {first component}
                   SEX    : Char;                    {second component}
                   WEIGHT : Real                     {last}
             End;                                    {end of record}
```

PERSON is a record object consisting of three components: AGE, SEX, and WEIGHT. The three components are nested within PERSON and are of differing types. Hence, PERSON is an object containing three values.

Examples of Records

The record structure is very powerful because it allows you to extend the simple types of Pascal to new data types which are closely related to your application. For example, if you are simulating a game of cards, the following data structure might be appropriate.

```
Type
    CARDS = Record
                  COLOR : Char;     {R = red; B = black}
                  SUIT  : String;   {Hearts, Clubs, Spades, Diamonds}
                  VALUE : Integer;  {face value or ?}
            End;
Var
    DECK : Array [1 .. 52] of CARDS;
```

The deck of playing cards is stored in an array called DECK. Each array element contains three values of differing types. Your program might shuffle the DECK, draw CARDS from DECK, and so on. The new data type called CARDS and the object named DECK closely resemble the actual objects being simulated.

As another familiar example, suppose the clock and calendar are simulated by two new record structures.

```
          Type
               TIME = Record
                          SEC : 0 .. 59;           {seconds}
                          MIN : 0 .. 59;           {minutes}
                          HR  : 0 .. 23            {hours}
                      End;
               DATES = Record
                          DAY : 1 .. 31;           {days of month}
                          MO  : 1 .. 12;           {12 mo/yr}
                          YR  : 1900 .. 2001       {year}
                       End;
               Var
                   CLOCK : TIME;
                   DATE  : DATES;
```

Record structures can be nested within record structures, as shown by this final example.

```
Type
    QUOTES = Record                                {stock market quotations}
                 Hi   : Real ;
                 Lo   : Real ;
                 Cloz : Real
                End;
Var
    Stock : Record
                Co    : String ;                   {company name}
                DATA  : Record
                            Q    : QUOTES;         {quotations}
                            Code : String;         {4 letter code}
                        End                        {inner record}
           End;                                    {outer record}
```

The structure of STOCK can best be described by a picture. In Figure 12.1, boxes Hi, Lo, and Cloz (containing real numbers) are inside of Q. Q and CODE are in an intermediate box called DATA. STOCK is in the outermost box and contains CO and DATA. Each record is a *component* of the enclosing record structure. The question which naturally arises is, "How are data entered and retrieved from these 'boxes'?"

Dot Notation

Pascal grammar uses the period or "dot" to denote a component of a record structure. Each dot corresponds with an enclosing box or record. For example, examine the following record structure:

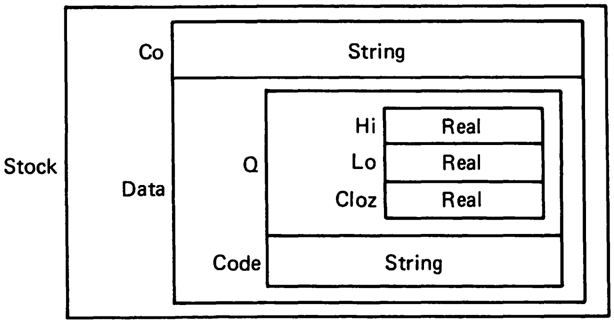


FIGURE 12.1
Structure of a nested record.

```
          Var
               BOX : Record
                          I  : Integer;
                          R  : Real;
                          C  : Char;
                          S  : String [10]
                      End;
```

The components of a BOX record are:

BOX.I	(integer component)
BOX.R	(real component)
BOX.C	(Char component)
BOX.S	(string component)

You assign a value to each component the same way you assign values to simple variables.

```
BOX.I   := 10;                   {assign component value}
ReadLn ( BOX.R );
BOX.S  := 'Hello, box';
```

The CLOCK declared earlier as a record structure containing seconds, minutes, and hours is manipulated like any other variable, except dots are used to qualify which component is referenced.

```
If (CLOCK.SEC + 1) > 59 Then
     begin
        CLOCK.SEC := 0;
        CLOCK.MIN := CLOCK.MIN + 1
     end;
```

This idea is used in the procedure shown in Figure 12.2 to simulate a 24-hour clock. In Procedure TIC_TOC, the simulated clock is advanced one second each time the procedure is called. This may cause the 24-hour clock to "roll over" to 0:0:0.

```
program Clock_Tic;
 type
  TIME = record
    SEC : 0..59; {seconds}
    MIN : 0..59; {minutes}
    HR : 0..23; {hours}
   end; {TIME}
 var
  CLOCK : TIME;
  Number, I : integer;  {working variables}

 procedure TIC_TOC (var TIMEX : TIME);
 begin
  if (TIMEX.SEC + 1) > 59 then
   if (TIMEX.MIN + 1) > 59 then
    if (TIMEX.HR + 1) > 23 then
     begin {reset to midnight}
      TIMEX.HR := 0; {time to go home}
      TIMEX.MIN := 0; {synchronize your watches}
      TIMEX.SEC := 0
     end {IF-HR-THEN}
    else
     begin  {BONG!  next hour starts}
      TIMEX.HR := TIMEX.HR + 1;
      TIMEX.MIN := 0; { new hr }
      TIMEX.SEC := 0  {seconds, too}
     end  {if-hr-else}
    else {middle if-else clause}
     begin  {TINKLE! next minute}
      TIMEX.MIN := TIMEX.MIN + 1; {same hour}
      TIMEX.SEC := 0; {new minute}
     end {middle if is done}
   else {outer if-else clause}
    begin {BEEP! next second}
     TIMEX.SEC := TIMEX.SEC + 1 {no problem}
    end {if mess}
  end; {TIC_TOC}
 begin
  Write('Enter current time: Hour=');
  ReadLn(CLOCK.HR);
  Write('Minutes=');
  ReadLn(CLOCK.MIN);
  Write('Seconds=');
```

FIGURE 12.2
A simulated clock. (continued on next page)

```
  ReadLn(CLOCK.SEC);
  Write('Enter number of tic_tocs ');
  ReadLn(Number);
  for I := 1 to Number do
   TIC_TOC(CLOCK);
  WriteLn('The correct time at the tone is ');
  Writeln(CLOCK.HR, CLOCK.MIN, CLOCK.SEC);
end. {Clock_Tic}
```

***FIGURE 12.2** (continued)*

Hands-On Experiment with TIC_TOC

Enter the entire program shown in Figure 12.2 and do the following:

1. Open WINDOWS-OBSERVE and enter the variables CLOCK.HR, CLOCK.MIN, and CLOCK.SEC as expressions to observe.
2. Click the Program Window, pull down and select the RUN-STOPS-IN item. Place a stop sign next to the TIC_TOC (CLOCK) statement, near the end of the main program.
3. Open the Text and Observe Windows so you can see their contents. You may have to move Observe to a location on the screen where you can see both the Observe and Text Windows at the same time.
4. Select RUN-GO. This causes the program to execute, pause, execute, and so forth, while pausing at the stop sign to update the values shown in the Observe Window.
5. Enter 23, 59, 55 as values for hour, minutes, and seconds. This is the current time. Enter 6 as the number of tics to process.
6. Watch the values in the Observe Window change as the clock ticks six times. The seconds will advance up to 59, then roll over to zero. The minutes and hours will also roll over to zero. The final value will be shown in the Text Window.

The action can be slowed down by selecting RUN-STEP over and over again. Each time the RUN-STEP selection is made, the Observer Window will show a new value for CLOCK. Watch as the CLOCK increases and then rolls over to midnight.

Nested Dots

Program Clock_Tic (Figure 12.2) shows how to access each component of a record structure, but what happens when the structure is several levels deep? A record inside a record is designated by two or more dots. Recall the structure in Figure 12.1. Here several dots are needed to resolve which value to access.

Notice how the dot notation mirrors the record structure; each dot corresponds to a level of nesting as does each Record-End in the declaration.

Here are *all* components of STOCK:

```
STOCK.CO
STOCK.DATA.Q.HI
STOCK.DATA.Q.LO
STOCK.DATA.Q.CLOZ
STOCK.DATA.CODE
```

The rule to remember is this: a dot notation is valid if it resolves an object *all the way down to a data type,* such as string, integer, char, Boolean, and real. A counter example, shown below, is *invalid* because it does not resolve to a data type.

```
STOCK.DATA.Q            ( invalid dot notation )
```

This will cause Macintosh Pascal to give you the thumbs-down error message "Types are not compatible" or something similar.

The With Statement

Dots can become cumbersome to use in a highly nested structure, so Pascal has a statement that allows you to drop the prefixes of dot notation variables. The *With* statement is a statement that works *only when the record structure prefix is referenced* and *not modified.* For example, PREFIX cannot be changed from within S (even though the other components of PREFIX can be modified from within S).

```
With PREFIX Do
    S;
```

The parts of this With statement are as follows:

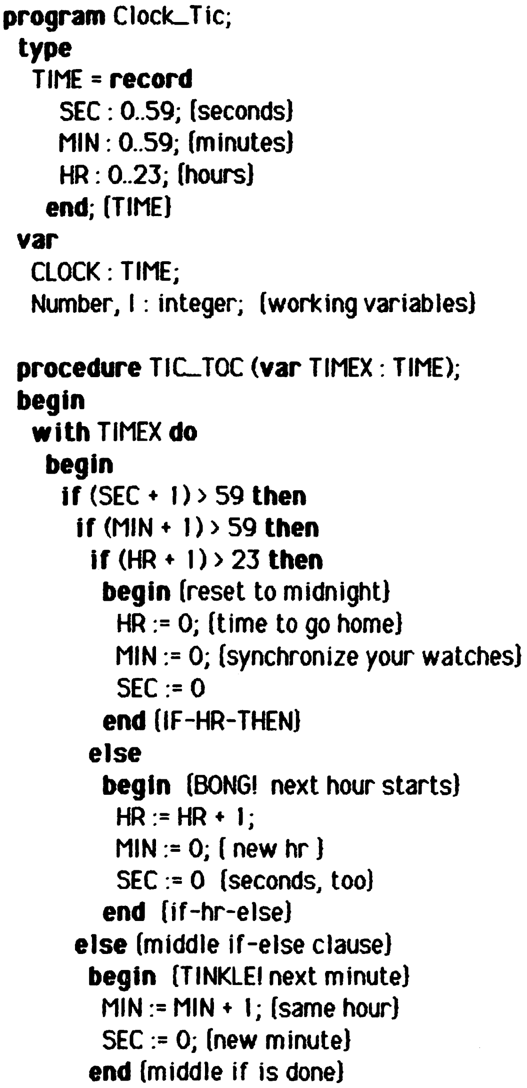
PREFIX is a dot notation expression (up to any level of nesting) which prefixes a record-structured variable.

S is any simple or compound statement.

For example, the modified CLOCK program of Figure 12.2 is shown in Figure 12.3. The dot notation variables CLOCK and TIMEX have been shortened using With statements.

***FIGURE 12.3** (continued)*

```
With TIMEX Do
   begin
              "
              "
              "
      SEC:=SEC + 1;          {use or modify a component of TIMEX}
              *
              "
   end;      (with)
```



```
program Clock_Tic;
 type
  TIME = record
     SEC : 0..59; {seconds}
     MIN : 0..59; {minutes}
     HR : 0..23; {hours}
    end; {TIME}
 var
  CLOCK : TIME;
  Number, I : integer;  {working variables}

 procedure TIC_TOC (var TIMEX : TIME);
 begin
  with TIMEX do
   begin
    if (SEC + 1) > 59 then
     if (MIN + 1) > 59 then
      if (HR + 1) > 23 then
       begin {reset to midnight}
        HR := 0; {time to go home}
        MIN := 0; {synchronize your watches}
        SEC := 0
       end {IF-HR-THEN}
      else
       begin  {BONG! next hour starts}
        HR := HR + 1;
        MIN := 0; { new hr }
        SEC := 0  {seconds, too}
       end  {if-hr-else}
     else {middle if-else clause}
      begin  {TINKLE! next minute}
       MIN := MIN + 1; {same hour}
       SEC := 0; {new minute}
      end {middle if is done}
```

FIGURE 12.3
CLOCK program using With statements. (continued on next page)

```
      else {outer if-else clause}
       begin {BEEP! next second}
        SEC := SEC + 1 {no problem}
       end {if mess}
     end  {WITH CLAUSE}
   end; {TIC_TOC}
  begin
   with CLOCK do
    begin
   Write('Enter current time: Hour=');
   ReadLn(HR);
   Write('Minutes=');
   ReadLn(MIN);
   Write('Seconds=');
   ReadLn(SEC);
   Write('Enter number of tic_tocs ');
   ReadLn(Number);
  end; {WITH}
 for I := 1 to Number do
  TIC_TOC(CLOCK);
 WriteLn('The correct time at the tone is ');
 with CLOCK do
  Writeln(HR, MIN, SEC);
end. {Clock_Tic}
```

Instead of writing the full name as a long dot notation, the With statement allows you to abbreviate all names using the prefix "TIMEX.". Similarly, "CLOCK." can be removed from expressions involving components, when the following With statement is used in the main program:

```
With CLOCK Do
```

In general, any level of prefixing can be taken care of using a With statement. For example, the nested STOCK variable could be handled by using a With as follows:

```
With STOCK.DATA.Q Do
 ReadLn (HI, LO, CLOZ);
```

Alternately, With statements can be nested to mirror the data structure being prefixed.

```
With STOCK Do
      With DATA Do
            With Q Do
                  ReadLn (HI, LO, CLOZ);
```

```
program Get_withit;
 type
  QUOTES = record
    HI : REAL;
    LO : REAL;
    CLOZ : REAL
   end;
 var
  STOCK : record
    CO : string;
    data : record
      Q : QUOTES;
      CODE : string;
     end
   end;
```

FIGURE 12.4
Nested With statements and dot notation. (continued on next page).

This form is not as readable, but is sometimes preferable when a program is working its way through different levels of a complex record structure.

Hands-On With Statements

A small experiment will help to familiarize you with the dot notation used in nested With statements. Start up Macintosh Pascal and enter the program shown in Figure 12.4. This program uses the STOCK variable we have been discussing. The purpose of Program Get_withit is to show you how to enter data into a record structure.

Select the Text and Observe Windows while running this program. Enter the following variables into the Observe Window expression boxes:

STOCK.CO
STOCK.DATA.Q.CODE
STOCK.DATA.Q.HI
STOCK.DATA.Q.LO
STOCK.DATA.Q.CLOZ

RUN-STEP-STEP and watch the values of each component of STOCK change as the following are entered:

Intel
64.5
30.25
35.75
INTC

```
begin
 with STOCK do
  begin
   Write('Enter company name ');
   ReadLn(CO);
   with DATA.Q do
    begin
     Write('Enter high ');
     ReadLn(HI);
     Write('Enter low ');
     ReadLn(LO);
     Write('Enter close ');
     ReadLn(CLOZ);
    end; {inner with}
   with DATA do
    begin
     Write('Enter company code ');
     ReadLn(CODE);
    end; {inner with}
  end; {outer with}
 WriteLn('Done')
end.
```

***FIGURE 12.4** (continued)*

Now, examine the structure of Program Get_withit more closely. The main program has three With statements: one outer With statement containing two inner With statements. The outer With statement establishes a prefix of "STOCK." for the inner statements.

The With DATA.Q Do statement establishes a prefix of "STOCK.DATA.Q." for all names contained within it. Thus, HI, LO, and CLOZ can be referenced without the need for lengthy dot notation.

Run this program again and try different input values and different expressions inside the Observe Window.

Summary

You should become familiar with record structures because they are used very often in Pascal programs. They improve readability and make programming easier and faster. Record structures can be declared in Type and Var statements, but not in Const or Function header statements. (Only single constants and base types apply to the Const and Function header declarations.)

Dot notation mirrors the record structure of a record structured variable. This is true consistently throughout Pascal. For example, an array of record structured elements is possible, as illustrated below.

```
Var
     A : Array [1 .. 10] of Record
                              X : Integer;
                              Y : Real;
                              Z : Char
                            End;                {record}
```

The dot notation is used exactly as you would expect:

```
A[5].X := 10;
A[3].Y := 3.2;
A[J].Z := 'm';
```

The indexes of an array appear as usual, and the components appear where you would expect them to appear. This consistency also holds for With statements:

```
With A[J] Do
     X := 3;
```

The With statement is simply a device for abbreviating names of record structured variables. With statements are used to access or "work through" a data structure without having to write long names and lots of dots. You should be careful when using With statements, however, because they can be confusing. (This becomes evident whenever you have two or more record structures interacting with one another in a program.)

Finally, you should note that it is possible to move entire record structures without reference to their components, if they are of the same types. Suppose, for example, P and Q are both of type TIME.

```
Var
     P, Q : TIME;
```

Instead of copying each component one at a time, you can simply transfer the entire record as follows:

```
P := Q;
```

This shortcut only works when the *entire* structure is copied from Q to P.

Problem Solving

1. Write a record structure for the following:
   a. A calendar with months (12), weeks, and days.
      A checkerboard with red and black squares.
   c. A complex (real and imaginary) number.
   d. A description of a person: weight, height, age, sex.
   e. An apartment to be rented.
   f. A student grade book.

2. Modify the TIC_TOC procedure and the program of Figure 12.3 so that it works on a 7-day week. Each day is stored as a string ('MONDAY', 'TUESDAY' . . . 'SUNDAY') in an Array [1 . . 7].
3. Write a dot notation for each component of the record structured variable described by the following piece of code:

```
With S Do
  begin
     A := 1;
     With T Do
       begin
          B := 'n';
          C := 2
       end;
     With W Do
          D := 5;
  end;
```

   Assume all identifiers are actually components of variable S.
4. Write a piece of Pascal code to display all components of the following structure, using With clauses:

```
Var
  A : Record
           X : Real;
           Y : Record
                 B : Integer;
                 C : Char
                End;
           Z : Char;
      End;
```

5. Use dot notation to list all components of the structure declared in Problem 4.
6. Write a procedure for entering a deck of cards into the following data structure:

```
         Type
           CARDS  = Record
           COLOR  : Char;
           SUIT   : String;
           VALUE  : Integer
         End;
Var
  DECK : Array [1 .. 52] of CARDS;
```

   Keep in mind that the format used in DECK [1]. COLOR is a valid dot notation for accessing a record-structured array element.

Session 13:

Sets and Scalars

In this session you will learn two new data types: sets and scalars. A set *is an unordered collection of manifest constants (a manifest constant is a* named *object whose value is fixed). A* scalar *is an ordered collection of manifest constants. Since these two are closely related, and yet often confused, you will learn their differences as well as how to use them.*

How Data Are Encoded

In 1799 a company of French soldiers (part of Napoleon's army) discovered a strange black monolith in a wall they were demolishing to make way for Fort Julien on the west bank of the Nile in Egypt. The large black basalt slab was covered with carved text, which turned out to be the same proclamation written in three different languages: Greek, demotic, and ancient hieroglyphic. The famous Rosetta Stone contained the key to understanding hieroglyphic writing, and through its decipherment, the ancient history of Egypt.

Learning the way computers work is much like deciphering an ancient code of writing. Hieroglyphs are icons of familiar objects such as trees, animals, and weapons. Data within a computer are coded representations of familiar objects such as real numbers, whole numbers, strings of characters, and so on. Some hieroglyphs directly represent the objects drawn (a human body), others stand for ideas suggested by these objects (a bull for power). Similarly, some data directly represent the values shown (52), while others stand for ideas suggested by the name of the value (NO_CARDS). When a name is used to stand for a value, we call this a *manifest constant.*

You have already used manifest constants. The Const statement declares integer, real, boolean, char, and string constants as shown below.

```
Const
     NO_CARDS = 52;                         {integer}
     INITIAL  = 'T';                        {Char}
     PROMPT   ='Enter Data ';               {string}
     PI       = 3.14159;                    {real}
```

This Const statement illustrates how to declare manifest constants whose types are familiar to you by now. With the exception of string, these are the *basic scalar types* in Pascal.

A *scalar* is a single value such as a number or character. A string, array, or record structure is *not* a scalar because each object of a string, record, or array type has more than a single value associated with its object.

Now that you are an experienced Pascal programmer, it is time to introduce you to two new data types which have a collection of manifest constants as their values. These are called *enumerated scalars* and *sets* in Pascal.

Enumerated Scalars

Suppose you want to associate the digits 0...9 with the names ZERO, ONE, TWO, THREE, . . . NINE in order to make the meaning of your program clear. The association shown below could be represented within a Pascal program in several ways.

0	1	2	3	4	5	6	7	8	9
ZERO	ONE	TWO	THREE	FOUR	FIVE	SIX	SEVEN	EIGHT	NINE

One method might employ Const and manifest constants, as shown below:

```
Const
     ZERO = 0;
     ONE  = 1;
      *
      *
      *
     NINE = 9;
```

You might also use a subrange type, but subrange would not allow you to use meaningful names in place of numbers.

An *enumerated scalar* type is an association between the integer subrange 0...n and the names ID0, ID1, ID2, . . . IDN. The association is made by simply listing the manifest constants in order, enclosed in parentheses.

```
Var
   NUMERALS : (ZERO, ONE, TWO, THREE, FOUR, FIVE,
                SIX, SEVEN, EIGHT, NINE);
```

The list is automatically associated with the whole numbers beginning with zero. To see this, do the following hands-on experiment.

Hands-On Scalars

Fire up your Macintosh Pascal and enter the following small program into the Program Window:

```
Program Scalar;
      Var
            COLORS : (RED, WHITE, BLUE);
Begin
            WriteLn (RED, WHITE, BLUE);
            WriteLn ( Ord(RED), Ord(WHITE), Ord(BLUE) )
End.           {scalar}
```

Select the Text Window and RUN-GO. The following output should appear in the Text Window:

```
redwhiteblue
       0      1      2
```

This is the correspondence between the manifest constants RED, WHITE, BLUE and the whole numbers 0, 1, 2. The correspondence *always* starts at zero and increases by one (the list is ordered).

Now modify the program by replacing the data declaration part with the following Type and Var statements and the program body with the following statements:

```
Type
      COLORS = (RED, WHITE, BLUE);           {new type}
Var
      COLOR, HUE : COLORS;                   {2 scalars}
      Begin
           ReadLn (COLOR);
           HUE := COLOR;
           WriteLn (HUE, Ord(HUE):3)
      End.
```

Select RUN-GO and type "WHITE" from the keyboard. Press RETURN and watch what appears on the next line. The manifest constant "WHITE" and its ordinal value are displayed.

Repeat the program by selecting RUN-GO a second time, only enter a non-valid manifest constant, say "GREEN". You should get a thumbs-down icon and a bug dialog that says,

```
"This is not in the enumeration list."
```

Only valid manifest constants are allowed as input (so be careful to spell them correctly).

Examples of Enumerated Scalars

Keep in mind that enumerated scalars are really manifest constants representing the subrange 0 .. N where (N + 1) names are in the list. They are like hieroglyphs which stand for the whole numbers they represent. You have already seen how the Ord function is used to get the underlying whole number represented by the manifest constant.

Enumerated scalars tend to make your program more reliable and much easier to understand. Here are some examples of their uses:

```
Type
     COLORS    = (RED, WHITE, BLUE);
     SEXES     = (MALE, FEMALE, OTHER);
     WEEKENDS = (SATURDAY, SUNDAY);
Var
     SUIT      : (CLUBS,  DIAMONDS, SPADES, HEARTS);
     SUMMER : (JUNE, JULY, AUGUST);
     YEAR      : (FROSH, SOPH, JR, SR);
```

Once an enumerated type has been declared, it and its subrange values can be used by another declarative statement. Here is an example of an enumerated subrange:

```
Type
     DAYS = (MON, TUE, WED, THUR, FRI, SAT, SUN);
Var
     WKEND : FRI .. SUN;
     WKDAY : MON .. FRI;
```

The only values accepted in WKEND are FRI, SAT, and SUN. If you attempt to assign anything else to WKEND, a bug dialog will get you.

Enumerated scalars are the names for whole numbers, so most operations that apply to integers also apply to integers also apply to scalars. Here are a few examples of how to process scalars:

```
        WKDAY := TUE;
        WKDAY := succ(WKDAY);           {next day}
        WKDAY := pred(WKDAY);           {backup a day}
        For WKDAY := MON To FRI Do      {count them}
            WriteLn (WKDAY);
```

Finally, scalars can be used anywhere whole numbers can be used, as shown by the following data structure for a simulated calendar:

```
    Type
        CENTURY = Record                                  {100 years}
                      YEAR   : 1900 .. 1999;              {20th century}
                      MONTH  : JAN .. DEC;                {from months}
                      DAYS   : 1 .. 31;                   {worst case}
                      CLOCK  : Record                     {wall clock}
                                  AMPM : (AM, PM);        {afternoon?}
                                  HRS  : 0 .. 12;         {include noon}
                                  MIN  : 0 .. 59;         {the usual}
                                  SEC  : 0 .. 59;         {ditto}
                               End
                  End;
```

This data structure assumes a previously declared enumeration:

```
        MONTHS = (JAN, FEB, MAR, APR, MAY, JUNE, JULY, AUG,
                  SEPT, OCT, NOV, DEC);
```

This defines a type whose subrange JAN . . DEC includes all of the MONTHS.

Enumerated scalars increase the readability of your program and make it more reliable by limiting the values that a scalar object can assume. They are not magical, however; when in doubt about when and how to use them, remember they are really the whole numbers 0 . . N.

Sets

Another kind of data structure commonly found in Pascal programs is the set. A *set* is an unordered list of manifest constants, but with a different collection of allowable operations than the operations permitted on enumerated scalars.

Think of a set as an object which can hold more than one element, much like an array. Most of the operations on sets are simple operations to put elements into a set, remove elements, and make comparisons to determine whether or not an element is in a certain set.

In Pascal a set is defined as a collection of ordinal values; integers, enumerated scalars, and characters may belong to a set, but reals, strings, and records cannot. The values allowed in a set are all the legal combinations of its scalars.

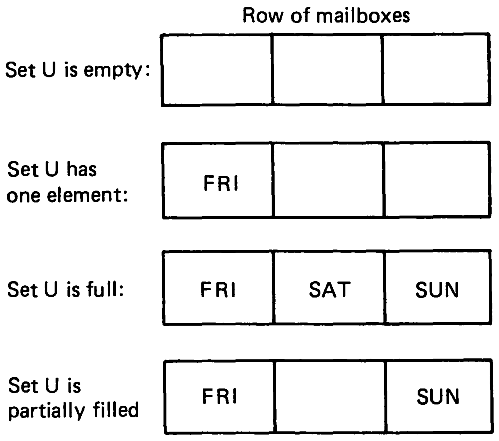


FIGURE 13.1
Sets as a row of mailboxes.

```
Var
   S : Set of 0 .. 1;
   T : Set of -1 .. 1;
   U : Set of (FRI, SAT, SUN);
```

S, T, and U can take on zero or more of the ordinals defined in their base type. The possible combinations of the subranges (without repetition) are:

S is empty, 0, 1, or 01

T is empty, -1, 0, +1, -10, -11, 01, -101

U is empty, FRI, SAT, SUN, FRI SAT, FRI SUN, SAT SUN, FRI SAT SUN

To keep sets separate from scalars and arrays, think of a set as a row of mailboxes. Each mailbox is a place to put a single letter. Each mailbox may be empty; if all the mailboxes are empty, then the entire set is empty (see Figure 13.1). Some of the mailboxes may be empty and some of them may be full. The entire collection of mailbox "values" constitutes the value of the set.

Examples Using Sets

Suppose the following declarations have been made prior to using sets:

```
Type
      WKEND = (FRI, SAT, SUN);
      GRADE = (A, B, C, D, F);
Var
      FUNDAYS : Set of WKEND;
      PASSING : Set of A .. C;
```

The following operations and their meaning should be studied before trying sets in Pascal:

```
FUNDAYS := [ ];                    {empty set}
```

The [ ] is called an *empty set*. The square brackets are the *set constructor* in Pascal; they cause a set to be constructed from the elements found within the brackets. Thus, to put one element into FUNDAYS, use the following assignment statement and set constructor:

```
FUNDAYS := [FRI];
```

Alternately, you could put all elements into a set by separating the list with commas:

```
FUNDAYS := [FRI, SAT, SUN];
```

A subrange of scalars can be put into a set using the subrange operator:

```
FUNDAYS := [FRI .. SUN];
```

Another way to add elements to a set is to include them using the *set inclusion operator* " + ":

```
FUNDAYS := FUNDAYS + [SAT];
```

Elements can be removed by subtracting them:

```
FUNDAYS := FUNDAYS - [FRI];
```

You can test for an element in a list by using the In operation, which returns a Boolean value:

```
If FRI In FUNDAYS Then
      Write ('Hurray')
Else
      Write ('Get down anyway!');
```

The set operators are listed below:

:=	Assign a set value to a set variable.
=	Compare two sets and return True if they contain the same elements.
< >	Return True if one set contains an element not found in the other set.
+	Add two sets by taking their union (no duplicates).
-	Remove the elements in one set from another set.
*	Form the intersection of two sets; all elements common to both sets are put in the resulting set.
<=	Return True if one set is completely included in another set, e.g., A< = B is
>=	True if A is completely contained in B.
in	Return True if an element is a member of a set.

The following operations are acceptable in Pascal:

```
PASSING := [A, B, C]          {include A, B, C}
PASSING := PASSING * [A];     {leaves A, only}
PASSING := PASSING - [A];     {now it is empty}
If C in PASSING Then
    WriteLn ('I passed')
ELSE
    WriteLn ('Oops. . .');    {is C in the set?}
```

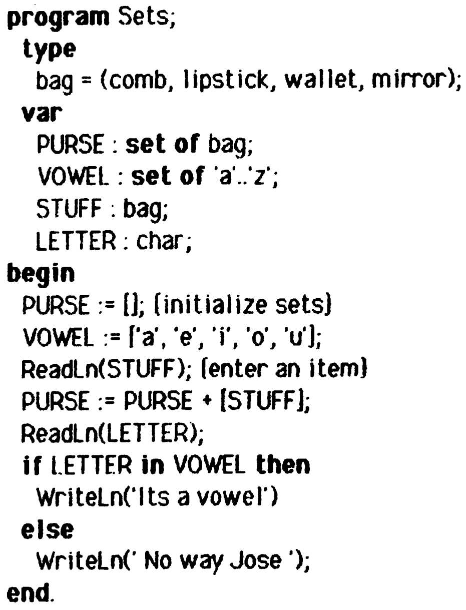
Hands-On Sets

The program in Figure 13.2 illustrates several important but subtle differences between sets, characters, and scalars. Enter this program into Macintosh Pascal and select RUN-STEP-STEP.

Here is a statement-by-statement explanation of Figure 13.2:

```
Type
   bag = (comb, lipstick, wallet, mirror);
```

This is a new enumerated scalar type with four manifest constants. This type will be used to form a set containing zero, one, two, . . . four elements.



```
program Sets;
 type
  bag = (comb, lipstick, wallet, mirror);
 var
  PURSE : set of bag;
  VOWEL : set of 'a'..'z';
  STUFF : bag;
  LETTER : char;
begin
 PURSE := []; {initialize sets}
 VOWEL := ['a', 'e', 'i', 'o', 'u'];
 ReadLn(STUFF); {enter an item}
 PURSE := PURSE + [STUFF];
 ReadLn(LETTER);
 if LETTER in VOWEL then
  WriteLn('Its a vowel')
 else
  WriteLn(' No way Jose ');
end.
```

FIGURE 13.2
Hands-on sets example.

```
Var
   PURSE  : Set of bag;
   VOWEL  : Set of 'a' .. 'z';
   STUFF  : bag;
   LETTER : Char;
```

PURSE is a set containing elements from bag. This declares the object, but does not put anything in it. The program body must fill PURSE with one or more elements. PURSE can be empty or contain one, two, . . . four elements.

VOWEL is a set which can be empty or contain one or more lower-case letters. The subrange 'a' . . 'z' includes all lower-case letters, but the set 'a' . . 'z' may contain zero, one, two, . . . twenty-six letters.

STUFF and LETTER are working variables whose types must match the base types of the sets they are going to work with. The base types of char and 'a' . . 'z' are both char, for example.

```
PURSE := [ ];
VOWEL := ['a', 'e', 'i', 'o', 'u'];
```

The two sets are initialized. PURSE is initially empty, and VOWEL initially contains the five lower-case letters given in the constructor list.

```
PURSE := PURSE + [STUFF];
```

The contents of STUFF are added to the empty PURSE. Now PURSE contains one element. The same result could have been achieved as follows:

```
PURSE := [STUFF];
```

But we could have put three elements in PURSE by using a loop:

```
For I := 1 To 3 Do
    begin
        ReadLn (STUFF);
        PURSE := PURSE + [STUFF]
    end;
```

A set cannot be written from a WriteLn statement, but you can list the contents of PURSE by indirectly retrieving and displaying its contents:

```
For STUFF := COMB To MIRROR Do
      If STUFF in PURSE Then
         WriteLn (STUFF);
```

Similarly, the PURSE can be emptied by subtracting elements, one at a time, from PURSE:

```
For STUFF := COMB To MIRROR Do
      If STUFF In PURSE Then
            PURSE := PURSE - [STUFF];
```

The elements of PURSE are manifest constants (enumerated scalars whose values are 0, 1, 2, 3, 4), but the elements of VOWEL are characters. You can add characters to VOWEL using the set constructor as follows:

```
VOWEL := VOWEL + ['y'];
```

Compare this method with

```
PURSE := PURSE + [COMB];
```

The character set requires single quotation marks around values, but the scalar set does not require quotes around a manifest constant.

Things to Remember About Sets and Scalars

If there are *N* elements in the base type of a set, then the set can hold up to 2^n different combinations of values. Thus, the set of (FRI, SAT, SUN) can hold eight different possible values. These values are the eight possible combinations of zero, one, two, and three values taken from (FRI, SAT, SUN).

Sets can be formed from scalars: integer, enumerated scalar, boolean, and char. Records, strings, and arrays cannot be the basis of a set variable. Subrange specification is allowed where meaningful, for instance, when characters are used.

```
  Var
        LETTERS : Set of 'A' .. 'Z';
Begin
        LETTERS := ['A' .. 'Z'];       (full set)
```

The set constructor is the pair of square brackets; it must be used whenever a set of values is being processed.

Sets cannot be written out to the Text Window, nor can they be entered from the keyboard. Enumerated scalars, on the other hand, can be both entered and displayed from a running program.

Elements can be added, subtracted, compared, unioned, intersected, and so on to a set only when the element is base-type-compatible with the set. (The base type of the element and the base type of the set must be the same.)

For, While, and Repeat loops can be used to process sets and scalars. The loop counter in a For loop can be an enumerated scalar, but not an element of a set. The "in" operator should be used to search a set for a particular element.

```
While PURSE <> [ ] Do
    begin
       ReadLn (STUFF);
       If STUFF in PURSE Then
            PURSE := PURSE - [STUFF];
       WriteLn (STUFF)
    end;
```

Sets are typically used, along with the In operator, to check for membership. For example, they are used to restrict user input, as shown below.

```
Repeat
     ReadLn (CH)                    {CH : char}
Until CH in ['0 .. '9', 'A' .. 'Z'];
```

Sets and scalars should not be confused. Sets are like a row of mailboxes, each box is either empty or contains one element. The entire row of boxes constitutes the set. A scalar, on the other hand, is a single value encoded as a manifest constant, integer, character, or boolean.

Summary

Use enumerated scalars to improve the readability of your programs. Use sets to simplify coding through the elegant application of set membership, comparisons, and so forth.

Sets cannot be either input or output via the keyboard and Text Window. They are used internally as a convenience.

Scalars, however, can be entered and displayed in Macintosh Pascal. This feature is nonstandard; if you want your programs to run on other computers, do not input/output enumerated scalars.

Scalars are manifest constants, and when used in an enumerated list, they become ordered. The Ord of a scalar yields its underlying value. You should be careful when mixing characters and scalars. Their base types are quite different, as illustrated by the two sets below.

```
['A', 'B']
[A, B]
```

The first set contains two characters, 'A' and 'B'. The second set contains two manifest constants from an enumerated scalar.

Problem Solving

1. Give the Boolean value which results from the following set operators:
   a. [1] = [1,2]
   b. 1 in [1,2]
   c. 0 in [ ]
   d. [0,1] <= [0, 1, 2]
   e. [0, 1] >= [1,2]
   f. 'A' in ['a' . . 'z']

2. Write a program similar to the program of Figure 13.2 which (1) allows the user to specify how many items will be put into PURSE, (2) loops to get the items from the user, and (3) then adds each item to PURSE.
3. Add the ability to display the elements of PURSE to the program in Problem 2.
4. Write a procedure that simulates a calendar. Each time the procedure is called from a main program, one additional day passes. Your procedure should handle the 12 months of the year, varying days, and so forth. Assume February has 28 days.
5. Design a data structure for a deck of playing cards and write a procedure for entering the deck of cards into a running Pascal program.
6. List all possible values for the following sets:
   a. Set of 1 . . 3;
   b. Set of (MALE, FEMALE);
   c. Set of '0' . . '2';
   d. Set of (RED, WHITE, BLUE);
7. Explain the difference between X and Y:

```
Var
   X : Set of (A, B, C);
   Y : A .. C;
```

Session 14:

Random and Typed Files

In this session you will learn how to use procedures Open, Close, Get, Put, Seek, and FilePos to process random files. In addition, you will find out how Write and Read can be used to output and input nontext data to a diskette file.

Nontext Files

A nontext file is a collection of records consisting of numbers, or numbers and characters mixed together. Nontext files are sometimes called *typed files* because the format of each record of the file is specified by a Pascal type.

The *"File of"* reserved phrase is used to declare a typed file. For example, a file consisting of records given by CLUMP would be declared as follows:

```
Type
     CLUMP = Record            {record format}
               X : Real;
               Y : Integer;
               Z : Char;
               S : String;
               End;     {CLUMP}
Var
     Ftype : File of CLUMP;    {internal filename}
```

Each record of the file called Ftype contains four components of differing types, hence the name "typed file."

Typed files must be opened and closed just like any other file, but the procedures chosen to do this depend on whether the file will be accessed sequentially or randomly.

Sequential vs. Random Files

Figure 14.1 shows the logical structure of file Ftype. Each record is numbered beginning with zero and ending with the LAST record. The end of the file is marked with an EOF character. Each time the file is accessed, an entire record containing X, Y, Z, and S is moved between main memory and the file on the diskette.

The file of Figure 14.1 can be processed sequentially by reading or writing the records in sequence (0, 1, 2, . . . LAST) or randomly by skipping around from one record to the other (5, 0, 1, LAST . . .). A randomly typed file has several advantages over a sequential file:

1. Random files can be accessed in a nonsequential manner. Thus, if only a few records are to be processed, it is not necessary to read the records in front of the records to be processed.
2. Typed files can store numbers and a mixture of numbers and text, thereby extending what can be stored beyond pure text files.
3. It is not necessary to rewrite the entire random file merely to change one or more records.

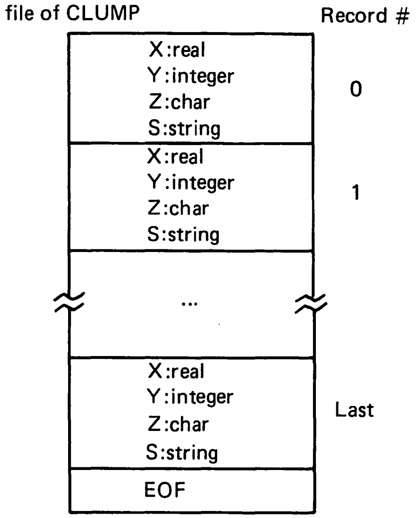


FIGURE 14.1
Logical structure of a typed file.

A random typed file also has some disadvantages, compared with text files:

1. The record type and its length must be fixed.
2. In order to retrieve a certain record from a random file, you must know the desired record number in advance.

In general, text and records that are subject to variable length are stored in sequential files; fixed-length records containing mixed data type components are stored in random files.

Random Files in Macintosh Pascal

A Macintosh Pascal random file is subject to additional constraints. Here is a brief list of things to remember about random files.

1. Create all files sequentially. When appending records to the end of an existing file, do so as a sequential file.
2. Read typed files using procedures for accessing random files, since random access is faster, in general.
3. Modify existing random file records by accessing them, making the modifications, and then putting them back in the file in the same location that they came from.
4. Record numbers should never be less than zero or greater than the last record number in the file.

Figure 14.2 shows how random file access works in conjunction with the *file buffer* pointed at by the internal filename.

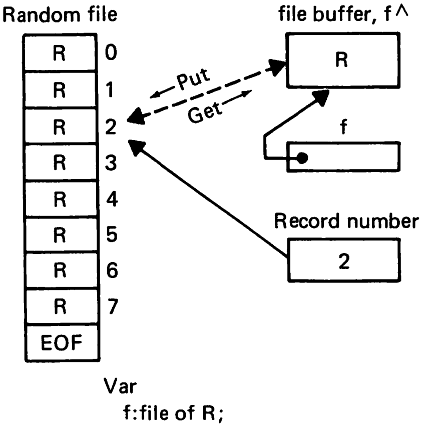


FIGURE 14.2
How a random file record is accessed.

f	Internal filename.
f^	File buffer for f.
Record number	Index of record to be accessed.
EOF	End of file character.

During a Read access, the record number is used to point to a record. The selected record is copied into the file buffer f^, so the program can access its components. Dot notation is used along with the up-caret (^) to designate components.

```
f^.component1
f^.component2
     "
     "
     "
f^.componentN
```

During a Write operation, the file buffer f^ must be filled with data (component-by-component), the record number selected, and the entire buffer copied to the corresponding file record.

All but the Seek operations on files automatically increment the record number by one either before or after an access. Automatic record number updating is one of the most confusing aspects of file access—so confusing, in fact, that it does not always work as it should even in Macintosh Pascal!

Constructing a Typed File

Now we can begin to write some useful routines for random file access. One constraint is that you build a new file sequentially. To do this, you will need the following tools:

Reset(f, Fname) Open sequential file for input.

Rewrite(f, Fname) Open sequential file for output.

Open(f, Fname) Open typed file f as Fname. If Fname already exists, the file is opened and the first record is read into main memory. Otherwise, a new typed file by the diskette directory name of Fname:string is created. The opened file may be either Written or Read.

Close(f) Close typed file f. This is the same as a text file Close.

Put(f) Write the file buffer contents f^ to the file at location Record Number (see Figure 14.2).

Get(f) Read the record at location specified by (Record Number + 1), into the file buffer of f^.

Seek (f, Record_No) Select record Record_No and prepare it to be accessed via the file buffer f^.

FilePos(f) :Integer Returns the current value of Record Number.

Read(f, list) Get values from the file and assign them to the list.

Write(f, list) Put the list values in the file record.

In addition to these definitions, you should memorize the effects each of the following have on the current record number:

Get Increments Record Number *before* reading a record into fˆ.
Put Writes the current contents of fˆ *before* incrementing the record number.
Seek Sets the record number.
Open Record number is set to zero.
Read Same as Get.
Write Same as Put.

Now suppose you want to create a file Fname of type Ftype:

```
        Type
            R = Record
                  X : Integer;
                  Y : String [10]
                End;
        Var
            Ftype  : File of R;
            Fname : String;
```

The following routine can be used:

```
Open (Ftype, Fname);                    {open typed file}
Repeat
    ENTER_DATA (Ftype);                 {put one or more records}
    Write ('More to enter (Y/N?');
    ReadLn (Answer)
Until (Answer = 'n') or (Answer = 'N');
Close (Ftype);                          {all done}
```

The ENTER_DATA procedure must be written and included along with this routine. The following simple ENTER_DATA routine shows how the records of Ftype might be entered, one at a time:

```
    Var
      Entry : R;                    {temporary}
Begin
    ReadLn (Entry.X);               {get components}
    ReadLn (Entry.Y);               {get components}
    Ftypeˆ  := Entry;               {copy to buffer}
    Put (Ftype)                     {write, increment record #}
End;
```

This procedure gets a record, moves it to the file buffer Ftypeˆ, and then forces the buffer to Write out to the file. The Open initially sets the record number to zero, so the first time a Put is performed, the zero-th record is filled *and then the record number is incremented* to 1. The next time Put is executed, the second record is copied to record 1, and the record number is incremented to 2. This continues sequentially until all records have been written to the newly created file.

The output was broken down into two steps to prove a point. First the file buffer must be filled, and then the buffer must be written to the diskette file.

```
Ftype^ := Entry;         {fill buffer}
Put (Ftype);             {write it}
```

This could have been done more directly with a single statement:

```
Write (Ftype, Entry);     {direct way}
```

In other words, a Write is equivalent to a copy followed by a Put. Notice the file buffer (Ftype^) is designated with an up-caret.

Reading a Typed File

Now let's see if we can do the reverse operation and Read the contents of a typed file back into main memory. The routine for doing this is simply:

```
Open (Ftype, Fname);          {set Record Number = 0}
Repeat
   WriteLn (Ftype^.X);        {display in Text Window}
   WriteLn (Ftype^.X);
   Get (Ftype)                {copy from buffer}
Until EOF (Ftype);            {all have been read}
Close (Ftype);
```

This routine may seem odd at first, because the Open procedure not only sets record number to zero, but it copies the zero-th record from the diskette into the file buffer Ftype^. Again, the Get is equivalent to the following pair of statements:

```
Read (Ftype, Entry);
Ftype^ := Entry;
```

Remember, the Get increments the record number *before* it copies a record into the file buffer. For this reason, the Get is done at the end of the Repeat-Until loop. The Get procedure advances the record number before fetching the next record to be displayed.

This method of processing a typed file is nearly identical to processing a sequential file. Once a file has been entered, it can be accessed randomly, using the Seek procedure.

Random Retrieval

An existing typed file can be opened and read randomly by using the Seek procedure followed by the Get procedure. Here is a routine to do this:

```
Open (Ftype, Fname);          {open typed file}
Read (Record_No);             {which record?}
```

```
Seek (Ftype, Record_No);      {Position record #}
Get (Ftype);                  {Get it}
WriteLn (Ftype^.x, Ftype^.Y); {show them}
Close (Ftype);
```

The seek procedure sets the record number to a certain value, then Get fetches the corresponding record. Do not Seek beyond the end of a file, and do not Seek negative record numbers! (According to the *Macintosh Pascal Reference Manual,* Seek is supposed to Seek and Get the record. We could not make this work, so to be sure, always use a Get after a Seek to guarantee that you get the desired record into memory.)

The nice thing about Seek is that you can jump around from one record to another without reading the records in between. In fact, you can update a record in the middle of a file by Seeking, Getting, editing it, Seeking again, and then putting the record back in the file.

Random File Update

A record in the middle of a typed file can be modified, if you are very careful how you do it. The idea is to get the record, modify it, and put it back in the same place in the diskette file.

The following steps should be followed when updating a typed file:

```
Seek (Ftype, Record_No);      {get a record}
Get (Ftype);                  {Make sure you have it}
Seek (Ftype, Record_No);      {Force Pascal to behave}
Ftype^.X =               ;    {changes. . .}
Ftype^.Y =               ;    {. . . changes}
Put (Ftype);                  {update record}
```

The Seek-Get combination gets the Record_No-th record from Ftype and stores a copy in Ftype^. The components in Ftype^ are modified; then Put moves a copy of the modified buffer back to the same record. Put also increments the current record number, thus making it point to the next record.

(Earlier versions of Macintosh Pascal did not work correctly. The Put would write the modified version of the buffer in the next record following Record_No. Even in the current version we had difficulty forcing the record numbers to behave when doing a Get operation. This problem can be overcome by Seeking before and after each Get just to make Macintosh Pascal remember where it is in the file. If you have an older version, trade it in on a corrected one.)

Hands-On Random Files

The ideas discussed in this session can be combined into a small mailing list program, as shown in Figure 14.3. Program Random_File should be entered into Macintosh Pascal and executed with the Observe Window containing File-

Pos (Ftype) and Record_No. Select RUN-STEP-STEP and watch both the Text and Observe Windows. The three procedures do the following things:

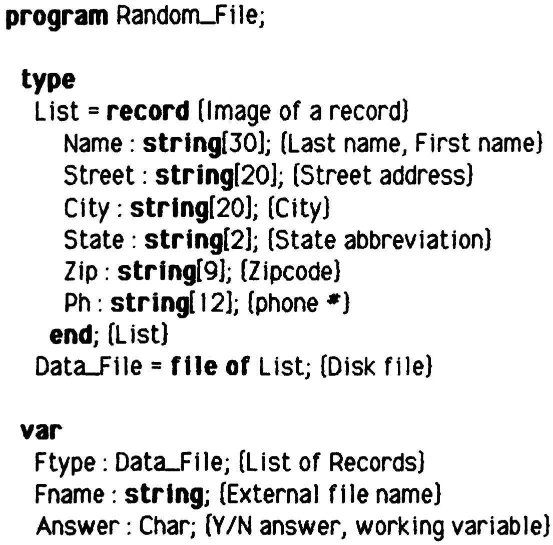
ENTER_DATA Captures name, street address, city, state, zipcode, and telephone number of a single person. Writes this information to the current record number of file Ftype.

LOOKUP_DATA Clears the screen and asks the user for a record number of a record to retrieve. Gets the record using Seek to position the record number to the desired record and copies that record into the memory buffer. Displays the contents of the buffer in the Text Window. LOOKUP_DATA displays one record.

DISPLAY_FILE Clears the Text Window and displays all records, beginning with record zero and ending with the last record. The Page intrinsic procedure clears the Text Window. The Get procedure reads the next record from disk.

The main part of this program Gets all records first, then displays the entire list, and finally lets you randomly select any record for display. Notice how the file name is obtained using the intrinsic function NewFileName and is then retained in variable Fname throughout the program.

Each time the file is Opened, its record number is set to zero. When the file already exists, record zero is read into the buffer by the Open procedure. (Actually, Open sometimes fails to pre-fetch the first record, so the Display section of the program in Figure 14.3 used two Seeks and Get to make Open work as advertised. Your version of Macintosh Pascal may not require this fix.)



```
program Random_File;

 type
  List = record {Image of a record}
     Name : string[30]; {Last name, First name}
     Street : string[20]; {Street address}
     City : string[20]; {City}
     State : string[2]; {State abbreviation}
     Zip : string[9]; {Zipcode}
     Ph : string[12]; {phone *}
   end; {List}
  Data_File = file of List; {Disk file}

 var
  Ftype : Data_File; {List of Records}
  Fname : string; {External file name}
  Answer : Char; {Y/N answer, working variable}
```

FIGURE 14.3
Random-file Mailing List program. (continued on next page)

```
procedure ENTER_DATA (var Ftype : Data_File);
 var
  Entry : List; {Data entry record}
begin
 with Entry do
  begin
   Page;
   Write('Name:');
   ReadLn(Name);
   Write('Street:');
   ReadLn(Street);
   Write('City:');
   ReadLn(City);
   Write('State:');
   ReadLn(State);
   Write('Zipcode:');
   ReadLn(Zip);
   Write('Phone #');
   ReadLn(Ph);
  end; {with}
 Ftype^ := Entry; {These two are..}
 Put(Ftype); {..same as Write(Ftype, Entry)}
end; {ENTER_DATA}

procedure LOOKUP_DATA (var Ftype : Data_File);
 var
  Record_No : integer; {Record number, working value}
begin
 Page;
 Write('Enter record number:');
 ReadLn(Record_No);
 Seek(Ftype, Record_No); {copy Record_No record into Ftype^}
 Get(Ftype);
 with Ftype^ do
  begin
   Page;
   WriteLn(Name);
   WriteLn(Street);
   Write(City);
   Writeln(', ', State);
   WriteLn(Ph)
  end  {with}
end; {LOOKUP_DATA}
```

***FIGURE 14.3** (continued)*

```
procedure DISPLAY_FILE (var Ftype : Data_File);
begin
  Page;
  repeat {until EOF}
    with Ftype^ do
      begin
        Page;
        WriteLn(Name);
        WriteLn(Street);
        Write(City);
        Writeln(', ', State);
        WriteLn(Ph)
      end; {with}
    Get(Ftype)  {same as Read(Ftype, Entry);Ftype^:=Entry;}
  until EOF(Ftype)
end; {DISPLAY_FILE}

      begin {Main}
        Fname := NewFileName('Enter file name:');

      {Enter Records, One At A Time}
        Open(Ftype, Fname);
        repeat {until done}
          ENTER_DATA(Ftype);
          Write('More to enter(Y/N)?');
          ReadLn(Answer);
        until (Answer = 'n') or (Answer = 'N');
        Close(Ftype);

      {Display entire file}
        Open(Ftype, Fname);
        Seek(Ftype, 0); {Force record 0 to...}
        Get(Ftype);        {...be copied to buffer}
        Seek(Ftype, 0);  {Reset record * to fix bug}
        DISPLAY_FILE(Ftype);
        Close(Ftype);

      {Randomly Retrieve Records}
        Open(Ftype, Fname);
        repeat {until done searching}
          LOOKUP_DATA(Ftype); {Search file, randomly}
          Write('More to lookup(Y/N)?');
          ReadLn(Answer);
        until (Answer = 'n') or (Answer = 'N');

      end.
```

FIGURE 14.3 (continued)

Summary

Typed files can be processed sequentially or randomly. A typed file has a certain type associated with its records. Here are several examples:

```
File of Integer;
File of Real;
File of Record
         X : String;
         Y : Real
      End;
File of Record
         M : Char;
         Z : Record
              S : String;
              T : Real
              End
      End;
```

Use the following intrinsic functions and procedures to process typed files (some of these have been discussed in the session on text files, but they also work in the expected way on typed files):

Reset	Open for input.
Rewrite	Open for output.
Open	Open for input and output.
Close	Close.
EOF	True if EOF reached.
GET	Advance Record Number and read.
Put	Write and advance Record Number.
Seek	Seek Record Number-th record.
FilePos	Return the current record number.

Read	Same as: Get (f);
	F_data := f^;

Write	Same as: f^ := f_data;
	Put (f);

NewFileName	Dialog box for getting a new filename.
OldFileName	Dialog box of getting an old filename.

Random file processing follows a pattern. Create and append new records to an existing file using sequential access methods. Look up and modify existing records using random access methods.

Problem Solving

1. What are the differences between the following:
   a. Random vs. sequential files?

b. Text vs. typed files?
c. Reset vs. Rewrite?
d. Seek vs. Get?
e. Write vs. Put?

2. Write a program to store a list of names, entered from the keyboard, in a file called PEOPLE. Add procedure that prints this list on the printer. Add another procedure that retrieves and displays any name, given its record number as input.
3. Modify the program of Figure 14.3 by adding a procedure called MODIFY_DATA which allows you to change all components of a randomly accessed record.
4. Suggest a data declaration section for a program that builds a file to keep student grades. The file must contain the following:
   a. Student's last and first names.
   b. Two midterm and one final test score (in percents).
   c. Five homework scores (in percents).
   d. Student's class standing (Fr., Soph., Jr., Sr.).
5. Write a program to create, display, and look up records in the file of Problem 4.
6. Modify the program of Figure 14.3 so you can search the data file for a certain name and display the entire record found in the Text Window. If the name cannot be found, your program should display, "Name not found."

Session 15:

More on Procedures and Functions

In this session you will learn about scope rules; nested procedures and functions; side-effects; global, local, and nonlocal names; recursion and the subtleties of advanced subprogramming techniques.

Scope Rules

Pascal is sometimes called a *block-structured* language because of the way Pascal programs are organized in units called *blocks.* A block is a section of program containing its own data declaration part and executable statement part. Const, Type, Var, and subprogram headings are put in the data declaration part, while executable statements are put between an enclosing Begin-End pair.

The main program consists of a block which in turn can contain other blocks. Procedures and functions are *named blocks*—they have an identifier and their own data declaration and executable statement parts.

Procedures and functions can be nested one within another; this causes many programmers difficulty in using Pascal. Nested blocks create program environments called *scopes.* The scope of an identifier (whether Const, Type, Var, Procedure, or Function) is the block in which it is declared.

Identifiers declared in the main program are called *global identifiers* because they exist in the outermost block and are accessible in all other parts of the program. The scope of a global variable, for instance, is the main program block.

Unfortunately, blocks can be nested within blocks as if they were boxes packed one inside of the other. Global identifiers penetrate these inner boxes, which gives rise to some problems to Pascal programmers. Some of the consequences of nested block structure are discussed in the remainder of this section.

Spelling Anomaly

A nested block may contain an identifier that is spelled the same as a global variable. The two names may look the same, but they are the names of two entirely different objects. When the inner name is used inside the inner block, the inner object is referenced. When the outer name is used inside the outer block (yet still outside the inner block), it references the outer object.

Here is a simple example showing that although objects X have the same name in two blocks, they refer to different objects.

```
Program ALIAS;
    Var
      X : Integer;          {global X}
Procedure INNER;
    Var
      X : Real;             {local X}
  begin
      X := 9.99             {inner object assignment}
  end;     {procedure}
Begin
    X := 100                {outer object assignment}
End. {ALIAS}
```

The inner X is different from the outer X. The inner X is called a *local identifier* and is used to access the local object. The outer or *global identifier* X cannot be accessed from within procedure INNER because its name is used to reference the local object.

The *most* local identifier always has precedence. In other words, if you make an assignment to a local identifier, it won't change the value of a global identifier with the same name.

Side-Effect Anomaly

A nested block may reference an outer object by using the name of the outer object from within the inner block. This is called a *side-effect* when an action from within the nested block produces a change to an outer block object.

```
Program SIDE;
    Var
      X : Integer;          {global X}
Procedure INNER;
    Var
      Y : Integer;          {local Y}
  begin
      X := 0                {side-effect}
  end;
Begin (SIDE)
    X := 1
    INNER:                  {side-effect}
    WriteLn (X)             {outputs: 0}
End. {SIDE}
```

This simple program illustrates how procedure INNER produces a side-effect in the main program. First, the value of X is set to 1; then INNER is executed. Within INNER, X is set to 0. The value of X becomes 0 in both the inner and global blocks.

Enter program SIDE in your Macintosh Pascal Program Window, then open the Observe Window. Enter variables X and Y as expressions in the Observe Window. Repeatedly select RUN-STEP and watch how variable Y goes from "Unknown" to some random (undefined) numeric value. X is set to 1 in the main program, and then to 0 in block INNER, as you STEP through the program.

Nonlocal Objects

The ramifications of block structure are shown in Figure 15.1, which matches the skeleton program shown in Figure 15.2. Put on your thinking headphones and follow this explanation:

First, program MAIN has three nested blocks inside its global block. The two blocks named B1 and B3 are nested one level deep. These two blocks are accessible by any statement in the executable statement part of MAIN.

Block B2 is nested within block B1—we say it is nested two levels deep. Procedure B2 cannot be accessed from the executable statements of MAIN because block B1 hides block B2 from view by the global block. The only way B2 can be executed is from a call within block B1.

The connecting lines in Figure 15.1 show the meaning of each identifier in terms of global, local, and nonlocal objects. A *nonlocal identifier* is the name of a nonlocal object. A *nonlocal object* is an object which is inherited from an outer, nonglobal block. For example, the objects from block B1 are also accessible to block B2, except for the Spelling Anomaly rule. This means all B1 locals are accessible by block B2, and all globals except the ones with identical spellings are accessible by B2.

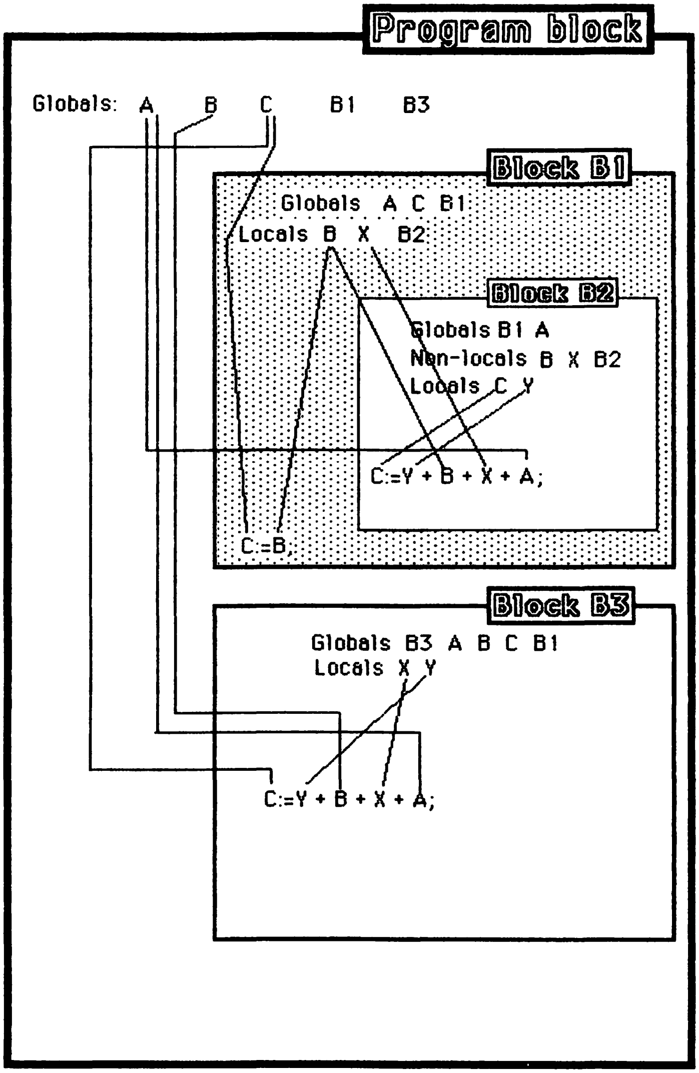


FIGURE 15.1
Scope rules.

Nonlocal identifiers are the names inherited from an outer, nonglobal block. A program must have at least two levels of nesting in order to have nonlocal identifiers.

```
program MAIN;
 var
  A, B, C : INTEGER; {Global variables}
 procedure B1;   {Global procedure}
  var
   B, X : integer;  {local variables}

  procedure B2;  {Nested local proc}
   var
    C, Y : integer; {local variables}
  begin {B2}
   C := Y + B + X + A; {uses scope rules}
  end;{B2}

 begin {B1}
  C := B;  {Side-effect}
 end; {B1}

 procedure B3;  {Another global proc}
  var
   X, Y : integer; {Locals in diff. block}
 begin {B3}
  C := Y + B + X + A; {Compare with B2}
 end; {B3}

begin {MAIN}
{Experiment here}
end.
```

FIGURE 15.2
MAIN program.

What Block Activation Does

A program block is activated whenever it is entered and its first executable statement is executed. Conversely, a program block is deactivated as soon as its last statement is executed and control leaves the block. A procedure is activated each time it is called and deactivated each time control passes back to the calling block.

Local objects do not exist unless the block containing them is activated. You can see a block become activated and then deactivated by running a program with a local variable displayed in the Observe Window. An object is "Unknown" if it is local to a deactivated block.

As soon as a block is entered, its local objects (Const, Type, Var) are created, but they remain undefined until values are provided for them. The Observe Window usually displays large negative numbers for undefined objects.

When a block is exited, all of its local variables no longer exist in main memory; hence, they become "Unknown." However, if a local identifier has the same spelling as a global identifier, the value of the name object will revert to the value of the global object. This effect will become apparent in the following hands-on experiments.

Hands-On Blocks

Perhaps the best way to become familiar with the scope rules of Pascal is through experience. In the following experiments you should use RUN-STEP and put in STOPS to make your program execute very slowly. Also, open the Observe Window and display all variables, so you can see them change values from one step to the next.

Modify the program of Figure 15.2 so that it contains executable statements, as shown in Figure 15.3. Each block contains a WriteLn to tell you it is activated. In addition, all local variables are assigned a value so you can follow what happens to them as the program is stepped to completion.

Initially, global variables A, B, and C are assigned the values of 100, 200, and 300. These values will change as blocks B1 and B3 are first activated (called) and then deactivated.

Variables A, B, and C are global variables that start out "Unknown," become undefined, and then finally become defined. Global variables B and C are spelled the same as local variables: B in B1 and C in B2. Hence, you will observe the Spelling Anomaly rule when running the program. Variable C is modified by a side-effect in B1 and another side-effect in block B3.

Next, modify Test1 by adding a call to Procedure B2 from within Procedure B1, as shown in Figure 15.4. Program Test2 calls Procedure B1, which in turn calls Procedure B2. RUN-STEP through the program and observe the changing values of all variables. Here is what happens when the assignment statement inside B2 is executed:

```
C := Y + B + X + A;          {uses scope rules}
```

Local object Y is referenced from within block B2 and added to nonlocal objects B and X referenced from block B1. Finally, the sum is increased by the amount stored in global object A referenced from the main program block. The following values are used to compute C:

```
program Test1;
 var
  A, B, C : INTEGER; {Global variables}
 procedure B1;    {Global procedure}
  var
   B, X : integer;  {local variables}

  procedure B2;  {Nested local proc}
   var
    C, Y : integer; {local variables}
  begin {B2}
   WriteLn('In B2');
   Y := 1;
   C := Y + B + X + A; {uses scope rules}
  end;{B2}

 begin {B1}
  WriteLn('In B1');
  X := 7; {define X, locally}
  B := 2; {define B, locally}
  C := B;  {Side-effect}
 end; {B1}

 procedure B3;  {Another global proc}
  var
   X, Y : integer; {Locals in diff. block}
 begin {B3}
  WriteLn('In B3');
  X := 3; {define X, locally}
  Y := 4; {define Y, locally}
  C := Y + B + X + A; {Compare with B2}
 end; {B3}

begin {MAIN}
 WriteLn('In Main');
 A := 100;
 B := 200;
 C := 300;
 B1;
 B3;
end.
```

FIGURE 15.3
Program Test1.

```
program Test2;
 var
  A, B, C : INTEGER; {Global variables}
 procedure B1;    {Global procedure}
  var
   B, X : integer;  {local variables}

  procedure B2;  {Nested local proc}
   var
    C, Y : integer; {local variables}
  begin {B2}
   WriteLn('In B2');
   Y := 1;
   C := Y + B + X + A; {uses scope rules}
  end;{B2}

 begin {B1}
  WriteLn('In B1');
  X := 7; {define X, locally}
  B := 2; {define B, locally}
  C := B;  {Side-effect}
  B2; {****Added to Test1****}
 end; {B1}

 procedure B3;  {Another global proc}
  var
   X, Y : integer; {Locals in diff. block}
 begin {B3}
  WriteLn('In B3');
  X := 3; {define X, locally}
  Y := 4; {define Y, locally}
  C := Y + B + X + A; {Compare with B2}
 end; {B3}

begin {MAIN}
 WriteLn('In Main');
 A := 100;
 B := 200;
 C := 300;
 B1;
 B3;
end.
```

FIGURE 15.4
Program Test2.

Y	is	1	from	B2
B	is	2	from	B1
X	is	7	from	B1
A	is	100	from	Test2

C is computed as 110, but then thrown away when B2 is deactivated. RUN-STEP through this example and convince yourself that the scope rules work.

Simple Scope Rules

Scope rules can be simplified if you never use the same identifier for two different objects in two different blocks.

1. The scope of an identifier is the block in which it is defined and all the blocks (nested or otherwise) contained within the definition block.
2. Global objects are declared in the outermost (main) block.
3. Local objects are declared in their own blocks.
4. Nonlocal objects are declared in some other block.

Some identifiers may not be accessible by blocks which do not contain them. This occurs whenever a nested block is contained within another nested block which shields the inner nested local identifiers from access. This is illustrated by Figure 15.1, showing local variables C and Y in B2, which cannot be accessed from statements in MAIN or B3. In fact, Procedure B2 cannot be called from anywhere other than block B1.

Parameter Passing

Passing information in and out of a block by side-effect is considered poor form by most programmers. Rather than commit such a *faux pas,* you should always communicate information from one block to another through parameter lists.

Recall that an actual parameter is an object listed in the procedure or function call. A formal parameter is an identifier listed in the procedure or function heading.

There are two ways to pass information back and forth between a calling block and a called block. The first method works for input to the called block; the second for both input and output to the called block.

Pass-by-Value

A parameter is passed-by-*value* if a copy of its value is made and the copy is assigned to the formal parameter at the time the called block is activated. The copy is destroyed when the called block is deactivated. Hence, this method works only when the parameter value is an input to the called block. "Pass-by-value" parameter passing has the virtue of causing *no side-effects.*

```
Procedure DESTINATION (A : Integer);
     begin
          WriteLn (A)
     end;
```

Pass-by-value parameters are declared in the procedure heading, as shown above. They become local objects when the procedure is activated; their initial value is obtained from the copy of the actual parameter's value.

```
Begin                      {MAIN}
     DESTINATION           {3 is the actual parameter's value}
End.                       {MAIN}
```

The actual parameter can be any expression which evaluates to a compatible type. The following actual parameters are computed before a copy is assigned to A:

```
DESTINATION (B * C);
DESTINATION ( Ord('Z') );
DESTINATION ( 3 * (B-5) );
```

Pass-by-Reference

The Var reserved word is used in the list of formal parameters to indicate that a parameter is to be passed by *indirect reference.* Instead of making a copy of the value to be passed, the rule of pass-by-reference is to pass a pointer to the value. A *pointer* is like a page number in the index of this book; it tells where a word or topic appears (its page number). The value of a pointer is the location in memory of the actual parameter.

A Var parameter is actually an alias for the original object. The value in the calling block is accessed and manipulated by the called block. No copies are made, and the value is changed as if by side-effect.

Pass-by-reference has two advantages:

1. Memory space is saved because no copy is made. This may be important when long arrays are passed.
2. The called block can return one or more computed values to the calling block.

There is one negative aspect to "pass-by-reference" parameter passing that you should be aware of:

> Actual parameters cannot be expressions or constants if they are passed by reference. Instead, they must be Var identifiers.

You can test this disadvantage by typing the following *incorrect* program in the Pascal Program Window:

```
Program MAIN;
    Procedure SUB (Var X : Integer);
        begin
        end;
Begin
  SUB (2)
End. {MAIN}
```

Selecting RUN-GO will result in a thumbs-down error message when the program reaches the call statement containing the constant 2 as an actual parameter.

External Blocks

An external block is one that is not contained within the global block of your program. There are two kinds of external blocks: (1) intrinsic functions and procedures and (2) user-defined functions and procedures.

Intrinsic functions and procedures consist of I/O subprograms such as WriteLn, Reset, Get, or Seek and special library functions and procedures given in the Macintosh Pascal manual. Library routines exist for doing sound, graphics, text processing, mouse I/O, and other Macintosh system operations.

User-defined functions and procedures are accessed through the "Uses" statement. (We will not discuss this statement here. See the Macintosh Pascal manual and the *Inside Macintosh* documentation for information on this advanced feature.)

Recursion

Consider the following definition:

> The sum of the first *N* natural numbers is equal to *N* plus the sum of the first *N*-1 numbers; the sum of the first number is 1.

To formulate this definition, assuming S denotes the sum, we can write:

```
1)    S(N) = N + S(N - 1)      {for the first clause}
2)    S(1) = 1                 {for the second clause}
```

Furthermore, let's compute the sum of the first five natural numbers, or in mathematical notation S(5). Substituting 5 for *N*, the first equation becomes:

$$S(5) = 5 + S(4)$$

You need the sum of the first four natural numbers before you can compute the value of S(5). Applying the same definition for S(4) produces:

$$S(4) = 4 + S(3)$$

Continuing in this manner yields the following:

```
(a) S(5) = 5 + S(4)
(b) S(4) = 4 + S(3)
(c) S(3) = 3 + S(2)
(d) S(2) = 2 + S(1)
(e) S(1) = 1          <--- according to the
                           second clause of
                           the definition
```

Careful examination of this list reveals that you can use the result of (e) S(1) to compute the result of (d), use (d) S(2) to find the result of (c), and so on through the list of formulas until you get the value for (a) S(5). Therefore, substituting for S(1), S(2), . . . until you reach S(5) should produce S(5) = 15.

Recursive Definition

This example is intended to show you an example of a definition (the sum of the first *N* natural numbers) used in its own definition. A definition in which the object being defined is used within its own definition is called a *recursive* definition.

Before going any further, let's analyze how the solution for S(5) was achieved. To arrive at the solution, we reduced the problem to the slightly simpler problem of finding the sum of the first four natural numbers and then added 5. This process was repeated over and over again, until we reached the self-defined value S(1) = 1.

In general, a recursive definition consists of two parts: One part relates the final solution to an intermediate solution of a simpler form of the same problem (i.e., S(5) = 5 + S(4)). The second part yields a nonrecursive solution to a specific segment of the problem (i.e., S(1) = 1).

Recursion in Pascal

Recursive solutions to programming problems are possible if procedures and functions are allowed to call themselves. In Pascal, when a procedure or function calls itself, it is called *simple* recursion. Also, a procedure or function may call a second procedure or function, which at some point calls the original procedure or function; this is called *indirect recursion.*

Each time a block is called recursively, all the local variables are copied; the new copies do not have any impact on the copies generated by previous calls. Therefore, if a procedure is called three times from itself, at some point four copies of its local objects exist (each object may possibly contain different values). This idea is important to understand, so it will be illustrated in the next experiment.

Hands-On Recursive Function S(*N*)

Fire up your Macintosh Pascal, enter Program SumRec, shown in Figure 15.5, and select RUN-GO. You should get 15—exactly the same result obtained when we analyzed S(5) earlier in this session. In Function Sum, the local value of *N* is compared with 1; if it is not equal to 1, Function Sum is called again. This process corresponds exactly to the first part of the recursive definition presented above. If *N* is equal to 1, Function Sum returns 1, corresponding to S(1) = 1 in the second part of the definition.

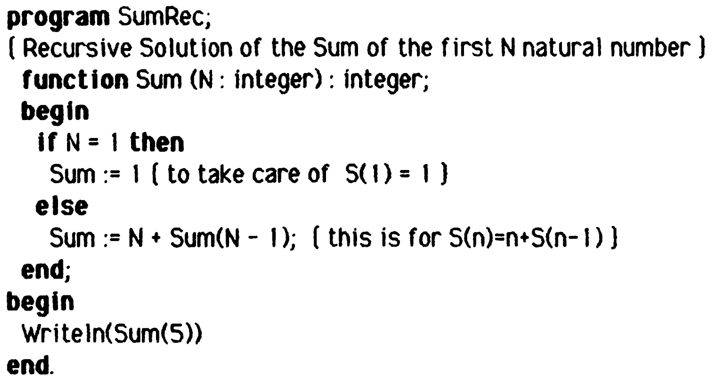
Now, to see how the function calls are performed, open the Observe Window and enter *N* as an expression. Insert a STOP sign in front of

```
If N = 1 Then
```

Select RUN-GO with the Observe Window visible so you can watch as the value of *N* changes. Notice that at first the value of *N* is 5. Select RUN-GO again. This time, the value displayed will be 4. As you repeat RUN-GO, *N* will decrease by 1 until *N* = 1; then the final result is computed and displayed in the Text Window.

Each of the values of *N* corresponds to one call of Function Sum. Also remember that for each call a copy of *N* is created (previous copies are not affected). What you saw in the Observe Window were the values of *different copies of N*.

Each time Sum returns from a call, its copy of *N* is destroyed. The copy of *N* = 1 is destroyed, followed by *N* = 2, *N* = 3 . . . until the final copy is destroyed and Sum returns 15 to the main program.



```
program SumRec;
{ Recursive Solution of the Sum of the first N natural number }
 function Sum (N : integer) : integer;
 begin
  if N = 1 then
   Sum := 1 { to take care of  S(1) = 1 }
  else
   Sum := N + Sum(N - 1);  { this is for S(n)=n+S(n-1) }
 end;
begin
 Writeln(Sum(5))
end.
```

FIGURE 15.5
Hands-on recursion.

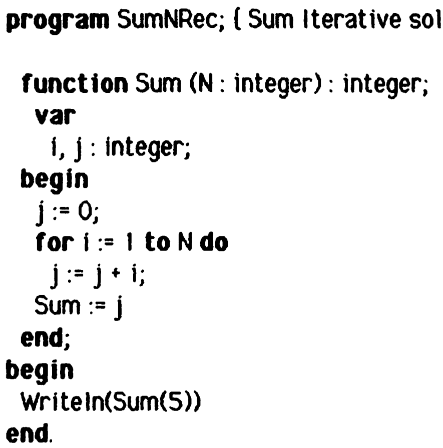
When Not to Use Recursion

Most of the time, a recursive solution is the most concise and elegant solution possible, but there are many occasions when recursive problems are best solved using simple iteration. Figure 15.6 is an alternate solution to the summation problem using iteration. Enter and run it, to see that you achieve the same result.

Often a recursive solution is not the best solution. As a simple rule, use a recursive solution if you need temporary storage at each stage of calculation; otherwise use an iterative method for solving the problem. In the hands-on example, there was no need to store temporary variables, so naturally the iterative solution is more attractive.

Forward Referencing

In Pascal, all identifiers should be defined before they are used; yet there are exceptions to this rule. One exception occurs in the definition of types, which you will learn about later. The second exception occurs when you must use procedures or functions before they are defined in the data declaration part of a block. In cases where it is necessary to use a procedure or function name before it is defined, the procedure or function heading should be declared as a *forward reference* (located before the procedure or function is called).



```
program SumNRec; { Sum Iterative solution }

 function Sum (N : integer) : integer;
  var
   i, j : integer;
 begin
  j := 0;
  for i := 1 to N do
   j := j + i;
  Sum := j
 end;
begin
 Writeln(Sum(5))
end.
```

FIGURE 15.6
Hands-on nonrecursive solution.

```
Procedure Add
      (Var Result : Integer; Num1 : Integer;
           Num2 : Integer ); Forward;              {forward reference}
```

A call to a procedure can be made even before the definition of its body appears. When Procedure Add is defined by forward reference, the actual procedure header appearing later *must not include its parameters.*

```
Procedure Add;               {notice, this time parameters are not repeated}
      begin
           Result := Num1 + Num2
      end;
```

This definition must appear at some point beyond the forward reference.

```
Program AVANTE;
   Procedure HAI (X : Integer); Forward;     {header only in reference}
   Procedure LOW;
      begin
           HAI (5)                           {reference to a forward}
      end;
   Procedure HAI;                            {no parameters here,
      begin                                   body definition only}
           WriteLn ('Hi!')
      end;
   Begin {AVANTE}
        LOW
   End.  {AVANTE}
```

Summary

Procedures and functions are valuable because they provide a mechanism for decomposing a large program into small, manageable parts. Divide-and-conquer is one of the most powerful methods of solving complex problems.

The scope rules of procedures and functions are sometimes confusing. There are two areas where they must be fully understood, however: nested blocks and parameter passing.

First, when in doubt, always use unambiguous names in nested blocks; this will help you avoid the Spelling Anomaly problem. Second, do not use global identifiers from within nested blocks; this will prevent side-effects.

In general, it is a good idea to avoid nesting beyond one level. Thus, a program should consist of a main block and many other blocks at the same level within the main program. This convention will guarantee that all subprograms are global and therefore can be called from anywhere in the main program or other subprograms.

Problem Solving

1. "Repair" Program Test2 by changing the names of identifiers that conflict with one another; then remove all side-effects. Your new program must use parameters to pass data between blocks.
2. Draw a diagram like the one in Figure 15.1 for the following program and its nested blocks.

```
Program MAIN;
    Var
        X : Integer;
    Function A (Y:Integer) : Integer;
          begin
              A := Y
          end:                  {function A}
  Procedure B (Var M : Integer);
        Var
            Z : Integer;
        begin
            Z := A(M)
        end;                    {procedure B}
Begin {MAIN}
      X := 0;
      B (X)
End.  {MAIN}
```

3. A function for generating random numbers is in an external procedure library called SANE. Run the following program and report the numbers generated.

```
Program RAND;
    Uses
        SANE;             {external blocks}
    Var
      S : Extended;       {extended arithmetic}
      i  : Integer;       {loop counter}
Begin
    S := 137;
    For i  := 1 To 7 Do
     begin
      S := Random;
      WriteLn(S)
     end
End. {RAND}
```

4. Modify the test program in Figure 15.4 as follows, then RUN-STEP to see what the effect is on the final value of C.
   (1) Remove local variable C from Block B2.
   (2) Remove the call to B3 from the main block so that B3 is not used.
   (3) Leave the statement C := Y + B + X + A; as it appears in B2.

   What is the effect on C?

5. Draw a diagram like the one in Figure 15.1 for the following program.

```
Program Test3;
      Var
           A : Integer;
Function T1 (A:Integer) : Integer;
      Var
           B : Integer;
      Function T2 (B:Integer) : Integer;
            Var
              C : Integer;
            begin
                 C  := 1;
                 T2 := T1 (C)
            end;      {T2}
      begin      {T1}
          B  := 2;
          T1 := B
      end;      {T1}
Begin {Test3}
           A := 3;
           A := T1 (A)
End.     {Test3}
```

6. Write a recursive function to compute the pi-product defined as follows:

```
pi = 1 * 2 * .. N              (where N > 0)
```

Recursively,

```
pi(N) = N * pi (N + 1)      (where N > 1)
pi(1) = 1                   (where N >= 1)
```

Session 16:

Pointers and Dynamic Data Structures

Data types such as arrays and records that you've learned about in earlier sessions are called static data structures *because their size remains fixed during program execution. In this session, you will learn about a class of data structures called* dynamic data structures. *A dynamic data structure can grow or shrink as a program runs.*

Why Do We Need Dynamic Data Structures?

Recall the program for building, sorting, and printing a list of phone numbers in Session 10. The solution given there stored the list in an array of a fixed size. You needed to know exactly how many phone numbers were to be kept in the list before declaring the array size accordingly. Rarely would you know in advance how many phone numbers to store. You can estimate the upper limit (say 100), but what if you only need to store a relatively small amount, say, five or ten numbers? In that case, your program would waste computer memory. Conversely, if the phone number list exceeds 100, the array would not be long enough and you'd need to modify the program to handle more numbers.

Addition Problem

When using an array to store a list you face the problem of how to do insertions (or additions) to the list. More specifically, if a list of phone numbers must be kept in order all the time, then adding a new number will cause one or more entries to be pushed down the list to make room for the new number in the correct position. For example, suppose a list has the following phone numbers ordered alphabetically:

Emily	305 452 4111
John	503 254 1114
Ted	350 542 1141

To add Molly 530 606 2128 in the right spot, you must move John and Ted to make room for Molly. This may not seem difficult when there are only two or three items to move, but what if there are 100 or more!

Deletion Problem

Similarly, when you delete items (a phone number, for instance) you must move all of the items below the deleted item up.

Examples like these show that static data structures are unsuitable for storing information whose length is subject to change. Dynamic data structures should be used to store information that may be changed through insertion, deletion, or addition.

Pointers

A *pointer* is another type like integer, real, or boolean. It is also called the *reference type* because pointers are used to reference objects indirectly. A pointer variable is a channel through which you can access other types of information. For example, a page number in an index containing important words in a textbook could be considered a pointer. To quickly locate the page containing information you can't recall, first look up the page number in the index and then turn to it. A page number is a reference to other information just as a Pascal pointer is a reference to other values stored in memory. In Pascal, pointer types are defined in the following way:

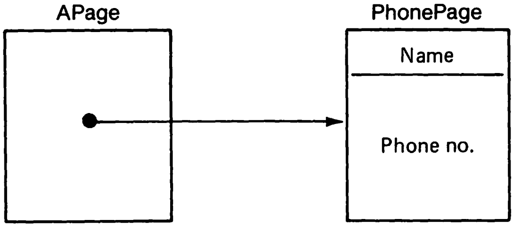


FIGURE 16.1
Pointer variable referencing an object.

```
Type
     PagePointer =^PhonePage;                {using the up-arrow}
```

You should read this type declaration as "PagePointer is a pointer to a PhonePage." The up-arrow or caret (^) symbol tells Pascal that PagePointer is a pointer type. What it means is that a variable of type PhonePage can be accessed with a pointer of type PagePointer.

In Figure 16.1, APage references an object of type PhonePage. Use APage^, which is read "APage points to . . . ", to show this association in Pascal. In general, the following rule applies:

APage: contains a pointer value (pointer). APage^: contains the value pointed at by pointer.

We often abuse the English language by calling APage a *pointer* and what APage^ points at the *pointee*, to keep the two separate.

Nil Pointers

Suppose you have a phone book and one of its pages is missing. Perhaps someone tore out the page that a friend's number is on. Now the references to the missing page are futile because they refer to "nothing." Similarly, in Pascal there is a special value for variables of type pointer called *Nil*, which points to *nothing*. For instance:

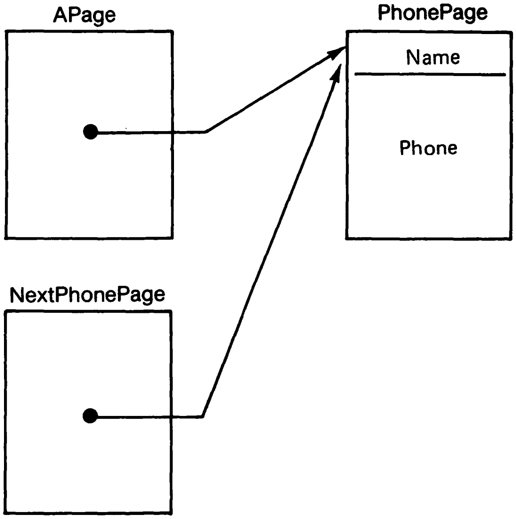


FIGURE 16.2
Two pointer variables referencing the same object.

```
APage := Nil;                {vacuum}
```

This means that APage is undefined. We usually use a special symbol such as the "ground wire" shown in Figure 16.3 to graphically represent a nil pointer.

Operations on Pointers

The operations allowed on pointers are:

:= Assigns a pointer to another pointer variable of the same type. For example, assuming both identifiers are of the same type:

```
NextPhonePage := APage;
```

This means that both NextPhonePage and APage point to the *same* object (see Figure 16.2).

= , <> Compares the value of two pointer variables for equality (=) or not equality (<>). Comparing for equality, the result will be True if both variables are pointing to the same object and False otherwise.

New (P) Creates an object of type P that can be referenced by P^. For example, to create a new PhonePage:

```
New (NewPhonePage) ;
```

Here NewPhonePage is a pointer to a PhonePage. To refer to an item in PhonePage:

```
NewPhonePage^.Name := 'Allen';
          or
NewPhonePage^.Phone := '555 777 5124';
```

Dispose (P) Destroys the object referenced by pointer variable P. For example, if NewPhonePage^ currently points to the PhonePage with Allen's number.

```
Dispose(NewPhonePage);
```

will destroy (omit, erase) the pointee information. NewPhonePage^ no longer designates anything.

Hands-On Linked Lists

Throughout this experiment, you will develop a simple list of phone numbers using dynamic data structures. Each time you add a new number to the list, a new page is created dynamically and *linked* or connected to a previous page. Imagine a phone book binder to which you add new pages each time you want to enter a new number. First, you will make a program to add new numbers to the front of the list. Then in the next section you will modify the program to make insertions into the middle of the phone book, as well as the beginning or end, so that it remains in alphabetical order.

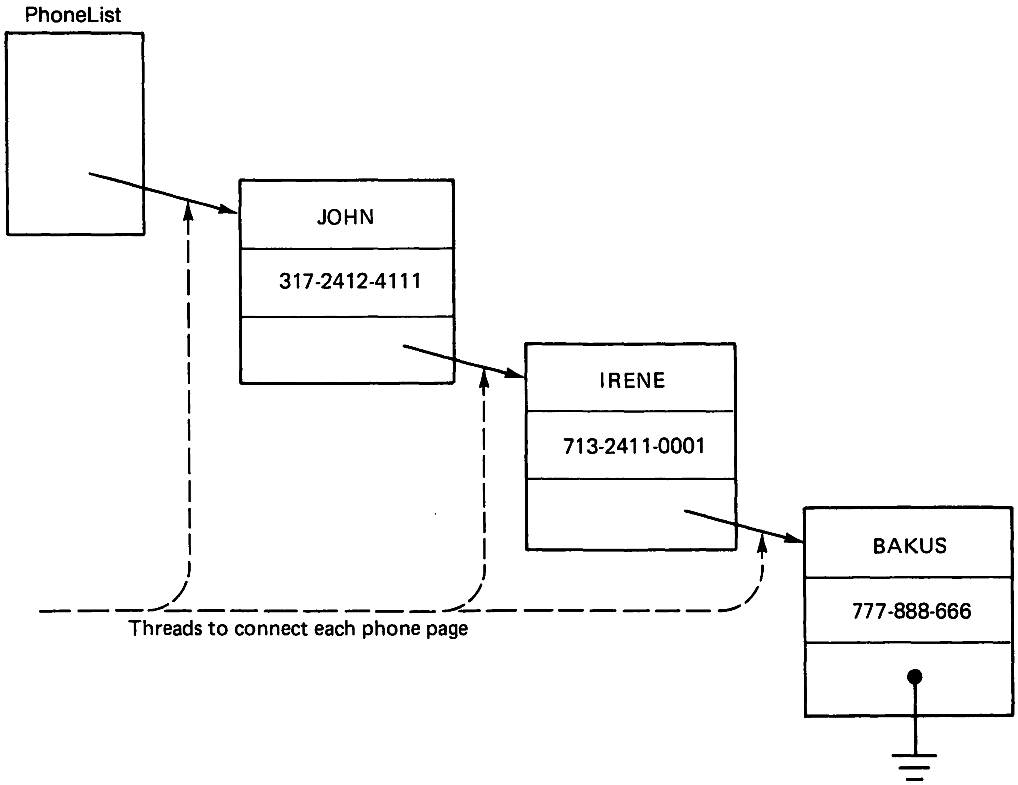


FIGURE 16.3
PhoneList with three phone pages.

The following type declarations and variables are needed, so bring up your Macintosh Pascal and enter:

```
Type
     PagePointer  = ^PhonePage;
     PhonePage    = Record
                    Name      : String [20];
                    Phone     : String [12];
                    Next Page : PagePointer
                    end;
     string20 = String [20];
     string12 = String [12];

Var
     PhoneList  : PagePointer;
     NextPhone  : Page Pointer;
     AName      : string20;
     Phone      : string12;
     done       : Boolean;
```

PhonePage defines the format of each page in the "binder." It defines a place for the name of the person (Name); a place for the phone number (Phone); and a place to be used for linking this page to the next page of the binder (NextPage), if there is a next page.

NextPage is a pointer variable of type PagePointer. Notice in the first type declaration that PagePointer points to a variable of type PhonePage.

```
PagePointer = ^PhonePage;           {points to PhonePage}
```

We've also defined the binder itself (PhoneList) as a pointer to the very first page of the list. PhoneList always points to the first PhonePage. See Figure 16.3 for the case of three phone pages in the PhoneList binder.

At the very beginning, the binder (or PhoneList) is empty, so it points to nothing.

```
PhoneList     Nil;
```

Adding to PhoneList

To add a new page to PhoneList, first create a new PhonePage. Second, record the information (name and phone number) and then add the new entry to the existing list.

```
New (NewPage);                      {creates a new PhonePage that
                                    can be referenced by NewPage}
NewPage^.Name     := 'Albert';
NewPage^.Phone    := '418 001 2781'
NewPage^.NextPage := Nil;
```

Notice how NextPage is set to Nil because (for the time being) it is not connected to any other page. Now add this new page to the PhoneList. For simplicity, let's add each new page to the beginning of the PhoneList and not worry about ordering them.

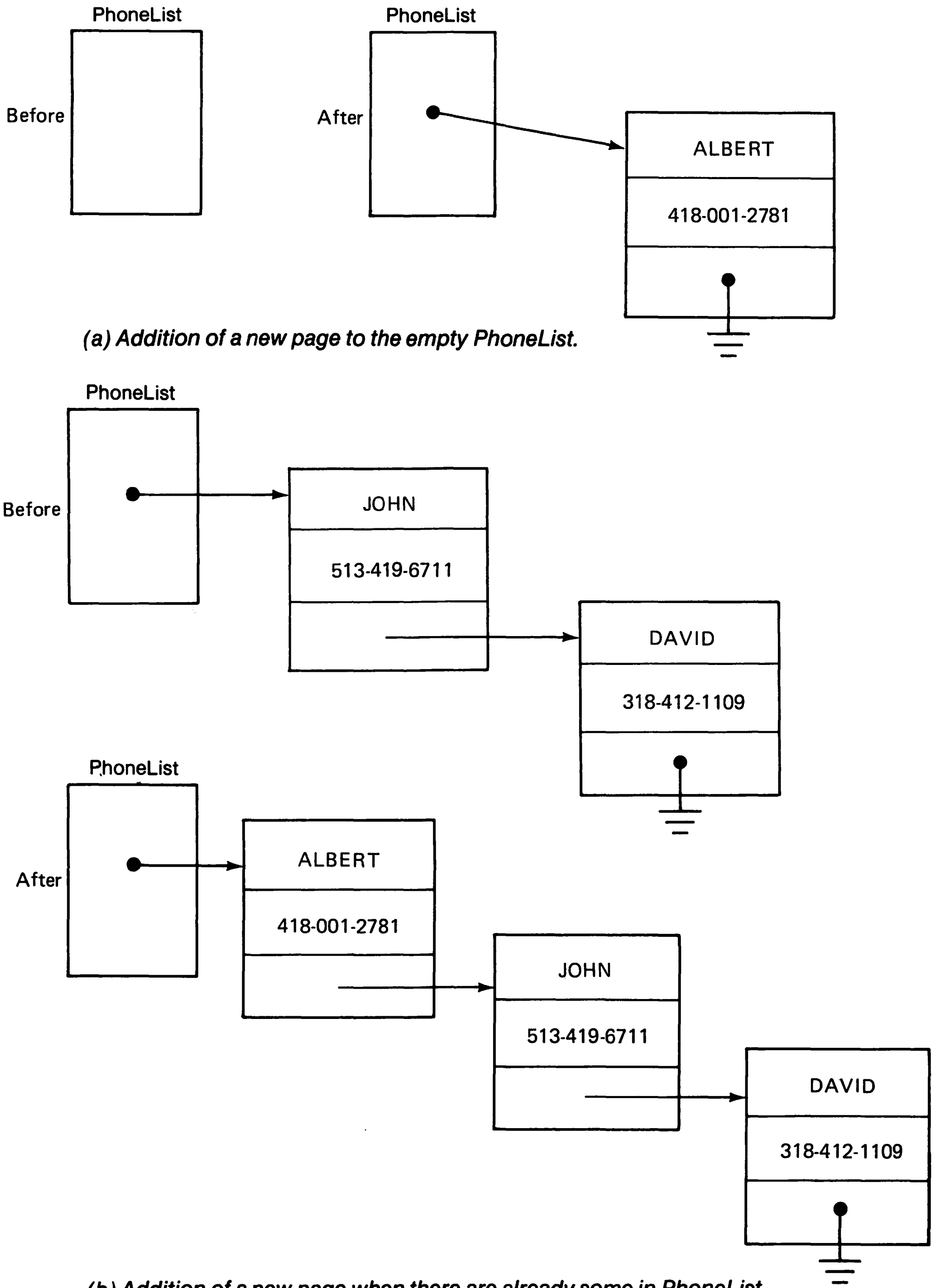
```
NewPage^NextPage := PhoneList;      {first connect existing pages to
                                    the new one}
PhoneList := NewPage;               {second, point to the new page}
```

This is how it works: The new page will be the very first page since PhoneList is initially Nil. The first step assigns Nil to NextPage, as shown in the "before" part of Figure 16.4 (a). If there are some pages already in the list, then the first step will insert a new page before the existing pages. (Refer to Figure 16.4 (b).)

Procedure AddPhone in Figure 16.5 employs the method explained above. AddPhone is called each time a new entry is added to the beginning of the list.

Printing a Phone List

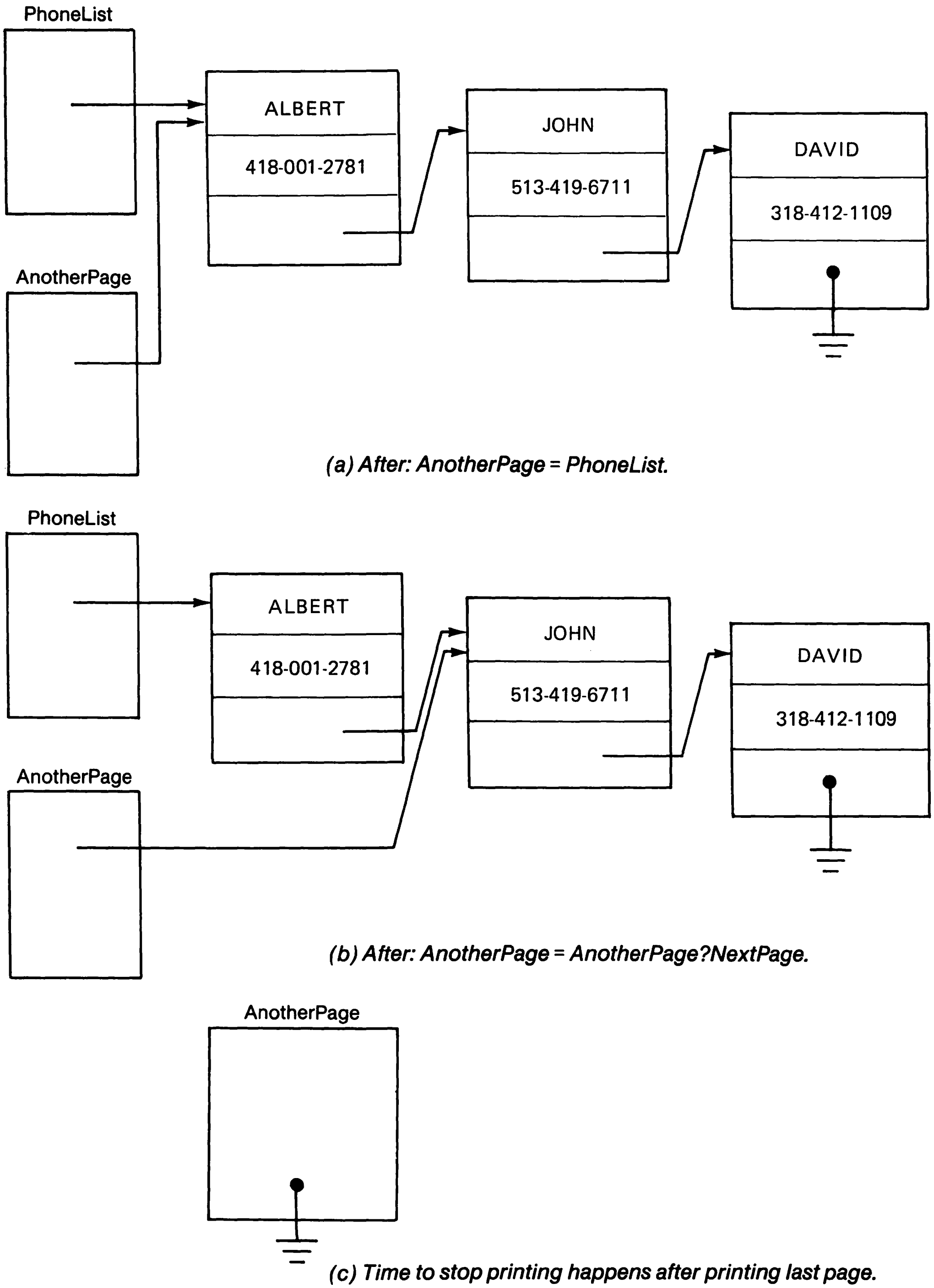
Procedure AddPhone takes care of building a list of names and phone numbers. To print or display each PhonePage, start from the very first page pointed at by



(a) Addition of a new page to the empty PhoneList.

(b) Addition of a new page when there are already some in PhoneList.

FIGURE 16.4



(a) After: AnotherPage = PhoneList.

(b) After: AnotherPage = AnotherPage?NextPage.

(c) Time to stop printing happens after printing last page.

FIGURE 16.5
Printing a list.

PhoneList, print it, and then use the pointer stored in NextPage to get the next page. Repeat the same action until the last page is reached. Since the pointer on the last page (see the page of David, Figure 16.4 (b)) is Nil, stop.

The following shows how to go from one page to the next (see Figure 16.5).

```
AnotherPage := PhoneList;                    {for first page}
AnotherPage := AnotherPage^.NextPage         {for going from one
                                             page to the next}
```

PrintPhones is a complete procedure to print the entire PhoneList:

```
Procedure PrintPhones (PhoneList : PagePointer);
  Var
    AnotherPage : PagePointer;
begin
   AnotherPage := PhoneList
   While (AnotherPage <> Nil) Do
       begin
          WriteLn ('Name : ', AnotherPage^.Name);
          WriteLn ('Phone : ', AnotherPage^.Phone);
          AnotherPage := AnotherPage^.NextPage
       end
end;                    {PrintPhones}
```

The While statement tests (AnotherPage < > Nil) for the end of the list; if so, the loop is terminated (AnotherPage is Nil).

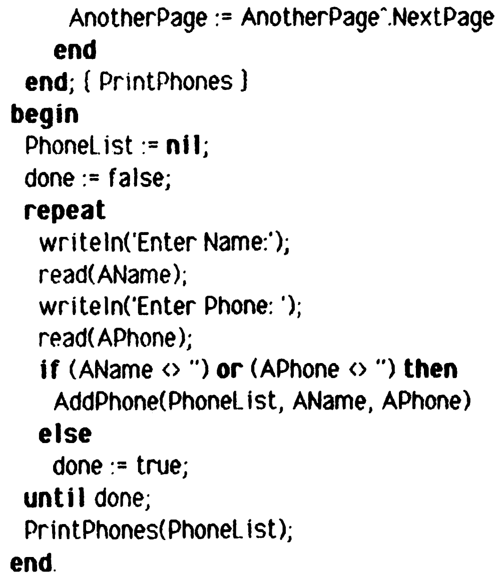
To make this hands-on example complete, you need a routine to read in names and phone numbers. The *Repeat loop* in the following code reads names and numbers and adds them to the list until an empty or Nil response is entered.

```
Begin
     PhoneList := Nil;
     done := false;
     Repeat
          WriteLn ('Enter Name: ');
          ReadLn (AName);
          WriteLn ('Enter Phone #: ');
          ReadLn (Aphone);
          If (AName <> ' ') or (APhone <> ' ') Then
             AddPhone (PhoneList, AName, APhone)
           Else
               done := true;
     Until done;
     PrintPhones (PhoneList);
End.
```

Figure 16.6 shows the complete listing of Program PhoneBook1. Type in the program and select RUN-GO to see how it works.

```
program PhoneBook1;
 type
  PagePointer = ^PhonePage;
  PhonePage = record
    Name : string[20];
    Phone : string[12];
    NextPage : PagePointer;
   end;
  string20 = string[20];
  string12 = string[12];
 var
  PhoneList : PagePointer;
  NextPhone : PagePointer;
  AName : string20;
  APhone : string12;
  done : boolean;
 procedure AddPhone (var PhoneList : PagePointer;
          Name : string20;
          Phone : string20);
  var
   NewPage : PagePointer;
 begin
  New(NewPage);
  NewPage^.Name := Name;
  NewPage^.Phone := Phone;
  NewPage^.NextPage := nil;
  if PhoneList = nil then
   PhoneList := NewPage
  else
   begin
    NewPage^.NextPage := PhoneList;
    PhoneList := NewPage
   end
 end;
 procedure PrintPhones (PhoneList : PagePointer);
  var
   AnotherPage : PagePointer;
 begin
  AnotherPage := PhoneList;
  while (ANotherPage <> nil) do
   begin
    Writeln('Name : ', AnotherPage^.Name);
    writeln('Phone : ', AnotherPage^.Phone);
```

FIGURE 16.6
Listing of Program PhoneBook1. (continued on next page)



```
    AnotherPage := AnotherPage^.NextPage
   end
 end; { PrintPhones }
begin
 PhoneList := nil;
 done := false;
 repeat
  writeln('Enter Name:');
  read(AName);
  writeln('Enter Phone: ');
  read(APhone);
  if (AName <> '') or (APhone <> '') then
   AddPhone(PhoneList, AName, APhone)
  else
   done := true;
 until done;
 PrintPhones(PhoneList);
end.
```

FIGURE 16.6 *(continued)*

Hands-On Linked Lists Revisited

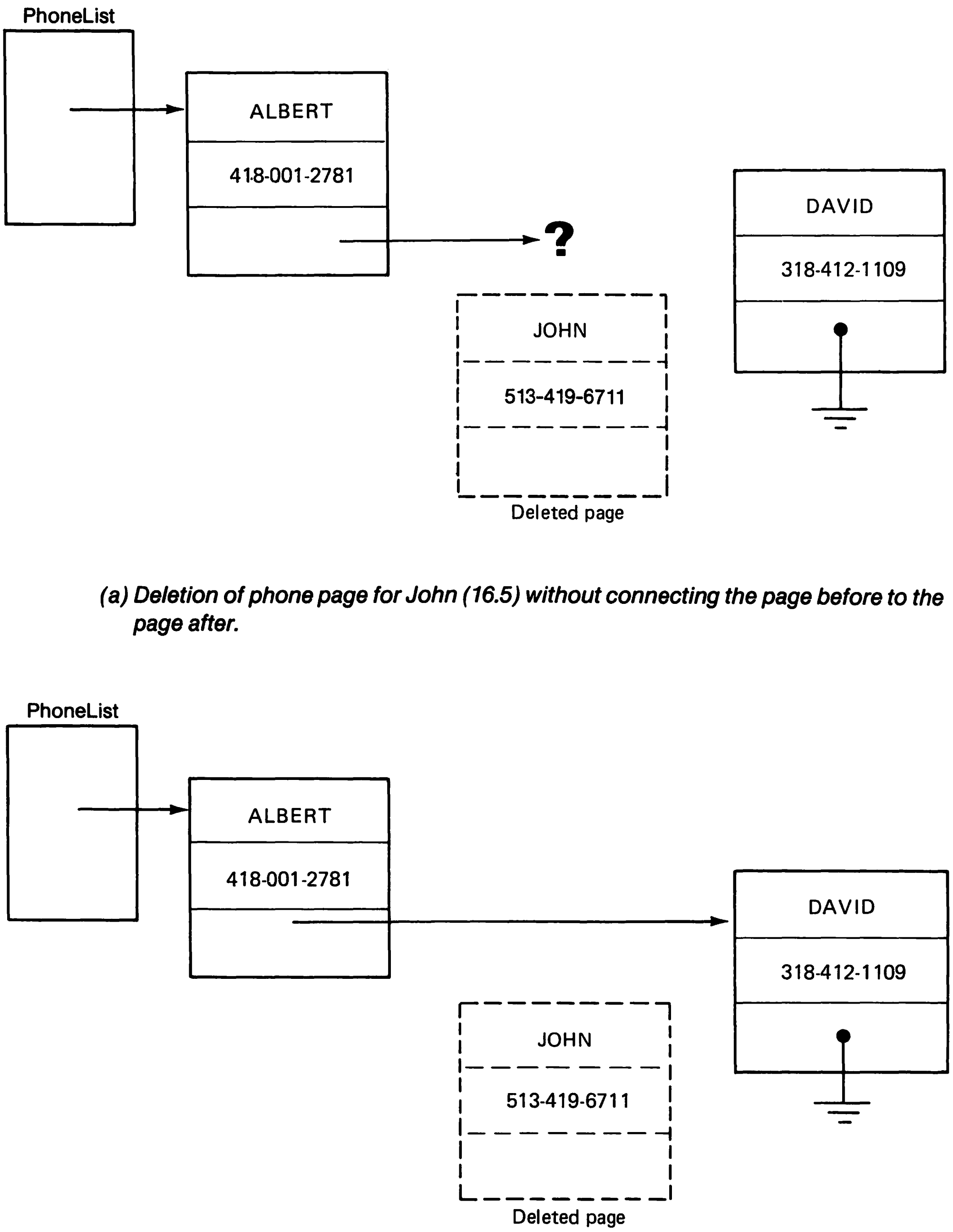
In this hands-on experiment, you are going to modify Program PhoneBook1 to make it more versatile. Basically, you will add a new function to delete an entry from the list and modify the procedure for inserting new phone numbers so that the PhoneList is always in alphabetical order.

Deleting from PhoneList

Two things must be done in order to delete a page from the list. First, locate the name to be deleted, and second, do the actual deleting.

Procedure FindPhone in Figure 16.9 locates a specific name and number. FindPhone starts from the beginning of the PhoneList and compares the names in each page with the name to be deleted. If the name is found, FindPhone returns the pointer value referencing the page (CurrentPage) and another pointer value referencing the page before the one to be deleted (PreviousPage). FindPhone also sets the Boolean variable "found" to true.

You must return the pointer to the page preceding the one to be deleted. Remember, we used the NextPage pointer variable stored in each page to make the connection to the next page in the list. Now if we delete the page containing the next pointer, the location of the next page will be lost! Furthermore, the list



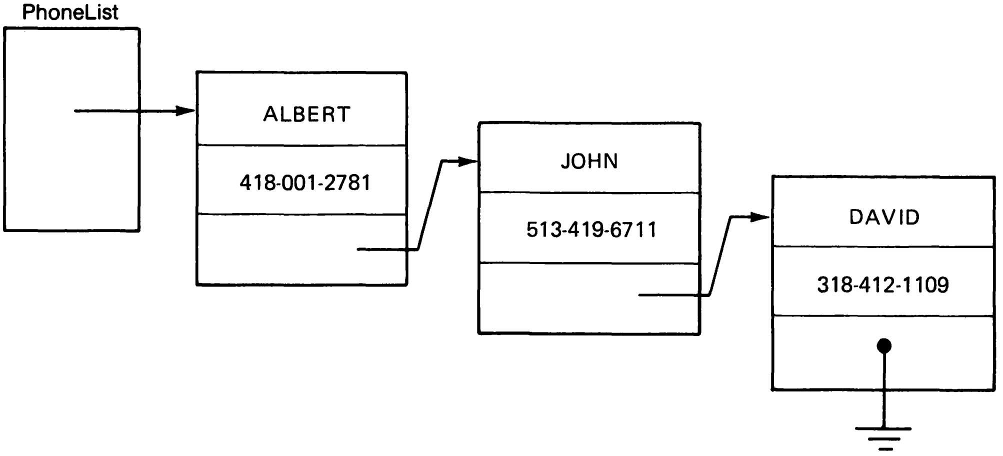
(a) Deletion of phone page for John (16.5) without connecting the page before to the page after.

(b) Deletion preserving the connection.

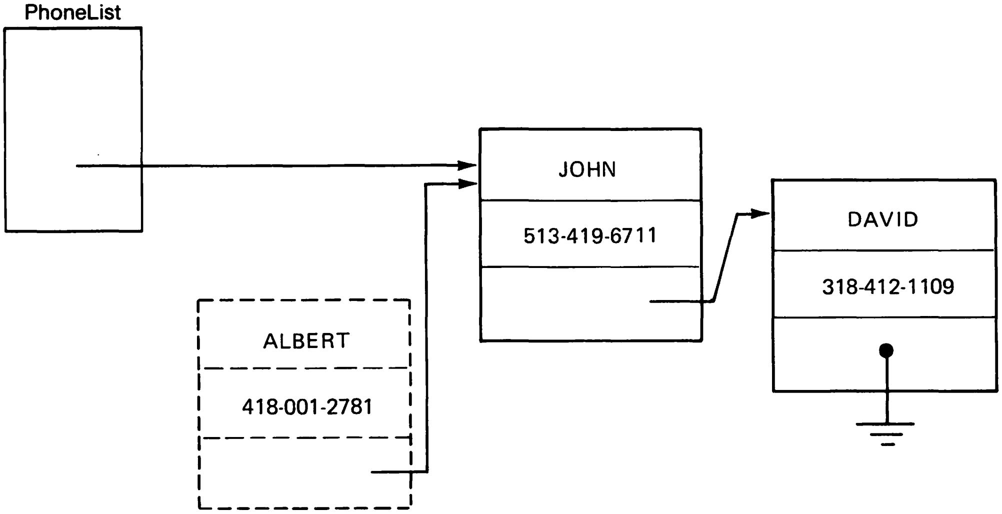
FIGURE 16.7

will "break" and your program will no longer be able to get to the page following the deleted one. See Figure 16.7 (a). To avoid a "break" in the list and preserve its continuity, make PreviousPage^. Next Page point to the page after the one to be deleted. See Figure 16.7 (b).

The pointer to the previous page and the pointer to the page to be deleted are both needed to connect the list again following a deletion.



(a) Before deleting the first page.



(b) After PhoneList = CurrentPage. NextPage, and before DISPOS (CurrentPage).

FIGURE 16.8
Deleting the first page of PhoneList.

```
                PreviousPage^.NextPage := CurrentPage^.NextPage;
```

Finally, to dispose of the page which was just removed, insert the following statement:

```
                Dispose (CurrentPage);              {release}
```

This will work fine as long as the deleted page is not the very first page; otherwise the value of PreviousPage is Nil. Add the If statement below to take care of this exception.

```
    If PreviousPage = Nil Then
        PhoneList := CurrentPage^.NextPage                 {if 1st page}
    Else
        PreviousPage^.NextPage := CurrentPage^.NextPage;
```

Figure 16.8 illustrates the deletion of the very first page of PhoneList.

Insertion in PhoneList

To insert new names and phone numbers in the proper locations and keep PhoneList in alphabetical order, we must almost duplicate FindPhone. First, search the list to find the right place to insert the new entry and then adjust the pointer values of the new page, the page before (PreviousPage), and the page after (CurrentPage).

When inserting a new element, three cases must be considered. First, the page insertion might occur at the very beginning of PhoneList because either the PhoneList is empty or NewPage should appear first alphabetically. Second, NewPage might be inserted at the very end of PhoneList and third, it might be inserted between two other pages. The following code handles all three of these conditions. For the complete program listing for insertion, see Program PhoneBook2 in Figure 16.9.

```
If found Then
  begin
    If PreviousPage = Nil Then
      begin
          NewPage^.NextPage := CurrentPage;         {insert at beginning}
          PhoneList := NewPage
      end
     Else
          begin
            NewPage^.NextPage := CurrentPage        {insert in middle}
            PreviousPage^.NextPage := NewPage;
          end
  end
Else
     PreviousPage^.NextPage := NewPage;              {insert at end}
```

The last change to PhoneBook1 makes it much easier to use. The following commands have been added to Program PhoneBook2:

S	for STOP.
A	for adding a new page to PhoneList.
D	for deleting a PhonePage.
P	for printing the PhoneList.

All commands can be entered as lower-case letters, as well as in capitals. After entering a command you will be prompted for more information if necessary.

```
program PhoneBook2;
 type
  string20 = string[20];
  string12 = string[12];
  PagePointer = ^PhonePage;
  PhonePage = record
    Name : string20;
    Phone : string12;
    NextPage : PagePointer;
   end;


 var
  PhoneList : PagePointer;
  NextPhone : PagePointer;
  AName : string20;
  APhone : string12;
  Command : char;
  done : boolean;
 procedure InsertPhone (var PhoneList : PagePointer;
          Name : string20;
          Phone : string12);
  var
   NewPage : PagePointer;
   PreviousPage, CurrentPage : PagePointer;
   found : boolean;
 begin
  New(NewPage);
  NewPage^.Name := Name;
  NewPage^.Phone := Phone;
  NewPage^.NextPage := nil;
  if PhoneList = nil then
   PhoneList := NewPage { The very first time }
  else
```

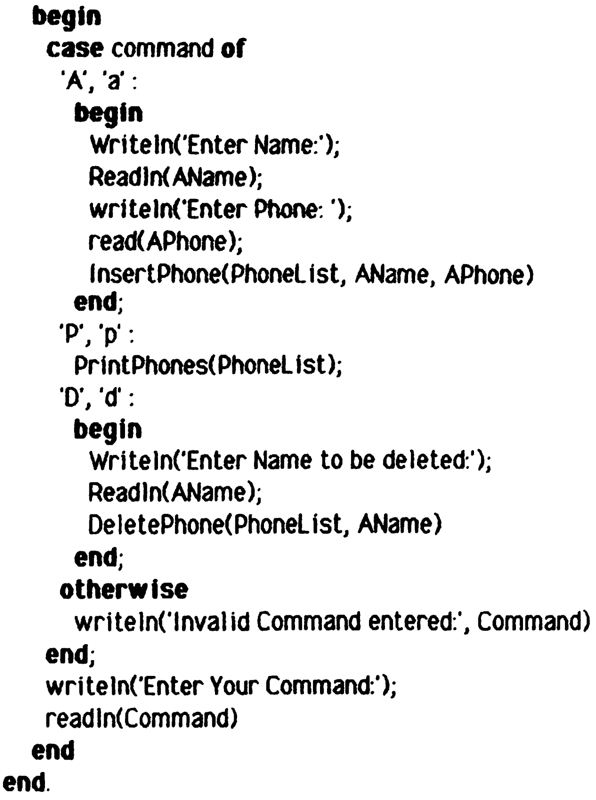
FIGURE 16.9
Program listing for PhoneBook2.

```
  begin
   PreviousPage := nil;
   CurrentPage := PhoneList;
   found := false;
   while (CurrentPage <> nil) and (not found) do
    begin
     if CurrentPage^.Name <= Name then
      begin
       PreviousPage := CurrentPage;
       CurrentPage := CurrentPage^.nextPage; { going to the next page}
      end
     else
      found := true
    end;
   if found then
    begin
     if PreviousPage = nil then
      begin
       NewPage^.NextPage := CurrentPage; { insertion at the beginning }
       PhoneList := NewPage
      end
     else
      begin
       NewPage^.NextPage := CurrentPage; { at the middle }
       PreviousPage^.nextPage := NewPage
      end
    end
   else
    PreviousPage^.NextPage := NewPage { end of list }
  end
end;
procedure FindPhone (PhoneList : PagePointer;
        Name : string20;
        var CurrentPage, PreviousPage : PagePointer;
        var found : boolean);
begin
 found := false;
 CurrentPage := PhoneList;
 PreviousPage := nil;
 while (CurrentPage <> nil) and (not found) do
  begin
   if CurrentPage^.Name = Name then
    found := true
   else
```

***FIGURE 16.9** (continued)*

```
    begin
     PreviousPage := CurrentPage;
     CurrentPage := CurrentPage^.NextPage
    end
  end;
end; { FindPhone }

procedure DeletePhone (var PhoneList : PagePointer;
        Name : String20);
  var
   CurrentPage, PreviousPage : PagePointer;
   found : boolean;
 begin
  FindPhone(PhoneList, Name, CurrentPage, PreviousPage, found);
  if found then
   begin
    if PreviousPage = nil then
     PhoneList := CurrentPage^.NextPage
    else
     PreviousPage^.NextPage := CurrentPage^.NextPage;
    Dispose(CurrentPage)
   end
  else
   writeln('Name:', Name, ' is not in the PhoneList')
 end; { Delete }


 procedure PrintPhones (PhoneList : PagePointer);
  var
   AnotherPage : PagePointer;
 begin
  AnotherPage := PhoneList;
  while (ANotherPage <> nil) do
   begin
    Writeln('Name : ', AnotherPage^.Name);
    writeln('Phone : ', AnotherPage^.Phone);
    AnotherPage := AnotherPage^.NextPage
   end
 end; { PrintPhones }
begin
 PhoneList := nil;
 writeln('Enter Your Command:');
 readln(Command);
 while (Command <> 'S') and (Command <> 's') do
```

FIGURE 16.9 (continued)



```
    begin
     case command of
      'A', 'a' :
       begin
        Writeln('Enter Name:');
        Readln(AName);
        writeln('Enter Phone: ');
        read(APhone);
        InsertPhone(PhoneList, AName, APhone)
       end;
      'P', 'p' :
       PrintPhones(PhoneList);
      'D', 'd' :
       begin
        Writeln('Enter Name to be deleted:');
        Readln(AName);
        DeletePhone(PhoneList, AName)
       end;
      otherwise
       writeln('Invalid Command entered:', Command)
     end;
     writeln('Enter Your Command:');
     readln(Command)
    end
  end.
```

FIGURE 16.9 *(continued)*

Summary

Use pointers to process lists which may vary in length or require insertion or deletion. Pointers are used to reference objects indirectly.

The up-arrow^ is used to designate a pointer value. Either A^ or ^A may be used. A^ means to reference the value that A *points to.* ^A means to declare a variable as a "pointer to A."

Linked list programs must be written carefully, because it is easy to change pointer values improperly. An improperly set pointer can point off into a vacuum and cause your program to crash. It is a good idea to draw a sketch as we have done here, to help visualize the "before" and "after" state of your list.

Pointers cannot be displayed or entered by way of Read or Write. They are for internal use by Macintosh Pascal. If you want to see a pointer, take up bird dog training.

Problem Solving

1. Add a new command "O" to PhoneBook2 which saves your PhoneList on a disk file as text.
2. Add a new command "R" to read the PhoneList from a text file and build PhoneList from it.
3. Run PhoneBook2, enter the following sequence of commands, and see what the results of each are:
   a. P
   b. a
      Steve
      513 315 1630
   c. a
      Anne
      303 417 2211
   d. p
   e. d
   f. M
   g. S
4. What change should be made to the insertion process in PhoneBook2 so that the PhoneList is built in descending order, instead of ascending order?
5. Write a program for maintaining an ordered list of names and addresses. Use your program to build a list of acquaintances.

Session 17:

Music (Sound)

In this session you will learn how to generate sound from a running Pascal program. You will use the sound generator intrinsic function StartSound and be introduced to elementary music theory.

Description of Music

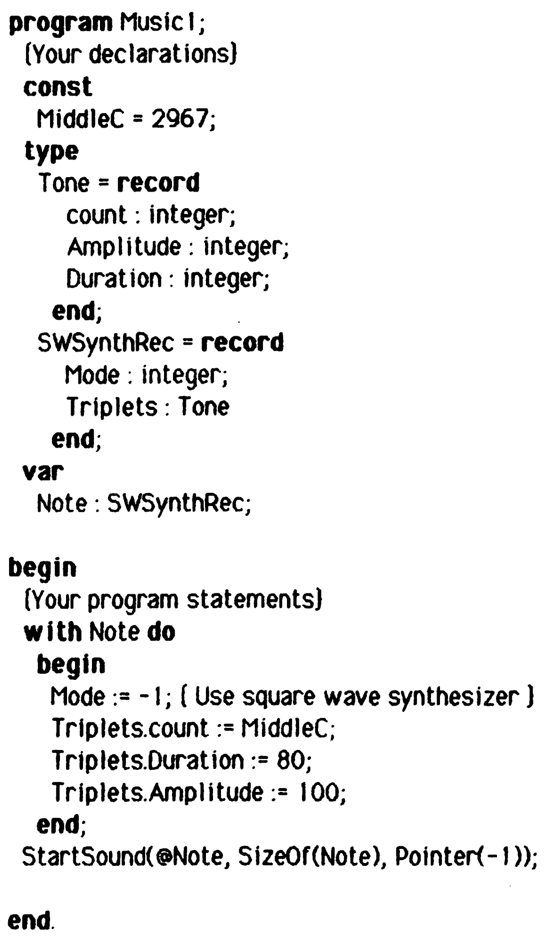
Mathematics is music for the mind;
Music is mathematics for the soul.

—Anonymous

If you happen to be among musicians and hear them use terms such as *bright, dark, hollow, harsh, golden, rich, raspy, woody,* and *reedy* to describe the quality of a tone, do not be surprised. They know exactly what they are talking about, because a musician's ear is trained to distinguish differences among various tones.

A computer cannot rely on subjective terms to describe the quality of sound it can produce. The only way you can describe tones to a computer is by using terms such as *frequency, intensity,* and *wave form.*

As an example, enter and RUN-GO the program in Figure 17.1. The sound you hear is the single note called middle C. Program Music1 describes middle C in terms of the frequency, duration, and amplitude of sound generated by the loudspeaker inside your Macintosh.



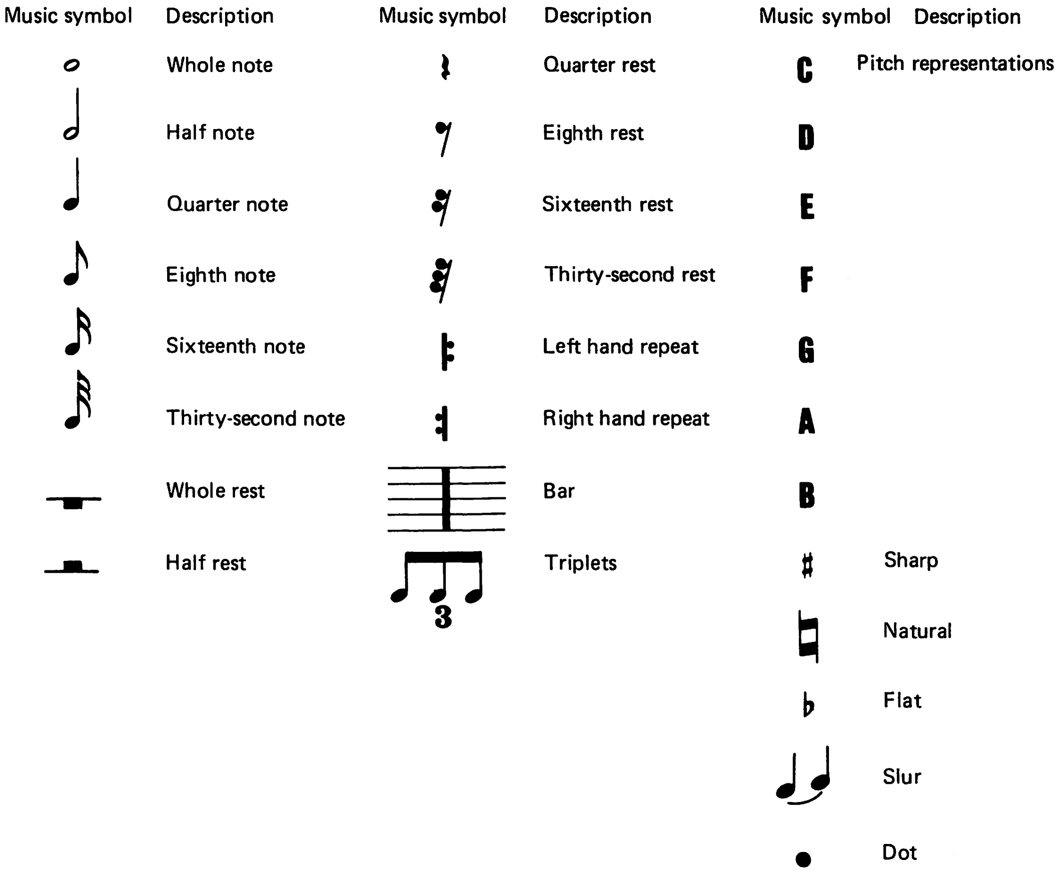
```
program Music1;
 {Your declarations}
 const
  MiddleC = 2967;
 type
  Tone = record
     count : integer;
     Amplitude : integer;
     Duration : integer;
    end;
  SWSynthRec = record
     Mode : integer;
     Triplets : Tone
    end;
 var
  Note : SWSynthRec;

begin
 {Your program statements}
 with Note do
  begin
   Mode := -1; { Use square wave synthesizer }
   Triplets.count := MiddleC;
   Triplets.Duration := 80;
   Triplets.Amplitude := 100;
  end;
 StartSound(@Note, SizeOf(Note), Pointer(-1));

end.
```

FIGURE 17.1
Music1.

These three attributes are defined by the integer values Count, Duration, and Amplitude. To see what these variables do, change their values to any other number between 0 and 255 and run Music1 again.

The Language of Music

Music has its own notation and grammar for communicating characteristics of sound. The table in Figure 17.2 lists the musical symbols and their descriptions. Figure 17.3 lists the meaning of each note in one octave of sheet music.



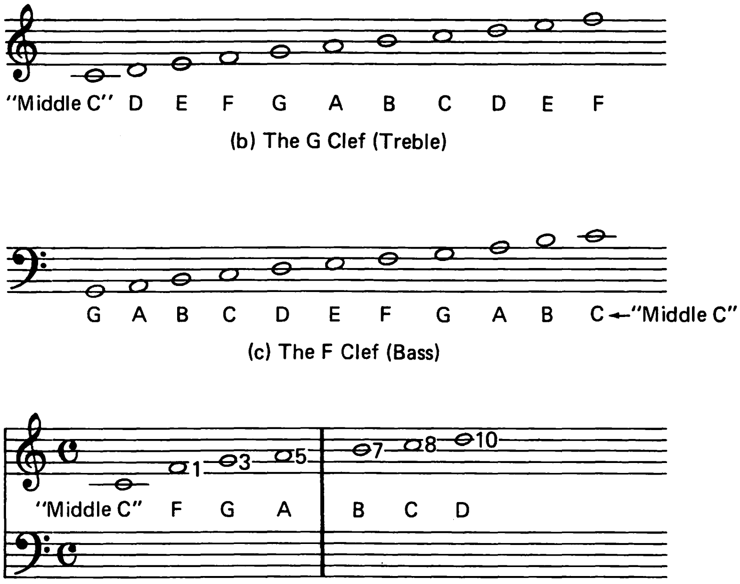
Music symbol	Description	Music symbol	Description	Music symbol	Description
𝅝	Whole note	𝄽	Quarter rest	C	Pitch representations
𝅗𝅥	Half note	𝄾	Eighth rest	D	
♩	Quarter note	𝄿	Sixteenth rest	E	
♪	Eighth note	𝅀	Thirty-second rest	F	
𝅘𝅥𝅯	Sixteenth note	𝄆	Left hand repeat	G	
𝅘𝅥𝅰	Thirty-second note	𝄇	Right hand repeat	A	
𝄻	Whole rest	𝄀	Bar	B	
𝄼	Half rest	♪♪♪ 3	Triplets	♯	Sharp
				♮	Natural
				♭	Flat
				♩‿♩	Slur
				●	Dot

FIGURE 17.2
Table of musical symbols.

Musical Scales

The source of sound is vibrating air, metal, plastic, strings, and so forth. A high-pitched sound is caused by rapid vibration; a low-pitched sound by slow vibration. The number of vibrations per second of sound wave, guitar string, or piano wire is called its *frequency*. Musical notes are defined by their frequencies, as shown in the table in Figure 17.4.

If the frequency of a note is doubled, we say it is an octave above the original note; if the frequency is cut in half, we say the note is an octave below it. The table in Figure 17.4 shows that the frequency of each higher octave note is twice the frequency of each lower octave note. As you move from one octave


"Middle C" D E F G A B C D E F
G A B C D E F G A B C←"Middle C"
1
3
5
7
8
10
"Middle C" F G A
B C D

(b) The G Clef (Treble)

(c) The F Clef (Bass)

(d) The Grand Staff

FIGURE 17.3
Musical notations.

FIRST	SECOND	THIRD	FOURTH	FIFTH	SIXTH	SEVENTH
C_1 33.0	C_2 66.0	C_3 132	C_4 264	C_5 528	C_6 1056	$C_7$2112
D_1 37.1	D_2 74.3	D_3 148.5	D_4 297	D_5 594	D_6 1188	$D_7$2376
E_1 41.3	E_2 82.5	E_3 165	E_4 330	E_5 660	E_6 1320	$E_7$2640
F_1 44.0	F_2 88.0	F_3 176	F_4 372	F_5 704	F_6 1408	$F_7$2816
G_1 49.5	G_2 99.0	G_3 198	G_4 396	G_5 792	G_6 1584	$G_7$3168
A_1 55.0	A_2 110.0	A_3 220	A_4 $\boxed{440}$	A_5 880	A_6 1760	$A_7$3520
B_1 61.9	B_2 123.8	B_3 247.5	B_4 495	B_5 990	B_6 1980	$B_7$3960
C_2 66.0	C_3 132.0	C_4 264	C_5 528	C_6 1056	C_7 2112	$C_8$4224

FIGURE 17.4
Table of frequencies (in Hertz) for seven octaves of the Just Diatonic Scale.

to another, the frequency changes by a factor of two. The Just Diatonic Scale is a musical scale in which octaves change by a factor of two. (Diatonic means "twice-the-tone.")

Sharp and Flat Notes

The difference between the frequency of any note and its sharp or flat can be calculated by the ratio 25/24 or 24/25, respectively. For example, the frequency of C sharp in the fourth octave can be calculated as shown below:

$$\begin{aligned}\text{CSharp} &= \text{frequency of } C_4 * 25/24\\ &= 264 * 25/24\\ &= 275\end{aligned}$$

To calculate the frequency of D_4 (D flat), use the following formula.

$$\begin{aligned}\text{DFlat} &= \text{frequency of } D_4 * 24/25\\ &= 297 * 24/25\\ &= \text{(almost) } 285\end{aligned}$$

Figure 17.5 displays the frequency of each note and its corresponding sharp and flat.

Hands-On: Playing All the Notes

Now let's modify Music1 to play all the notes on the fourth octave. First, you must calculate the count value related to the frequency of each note.

Count = 783360 / frequency of the note (from Figure 17.4)

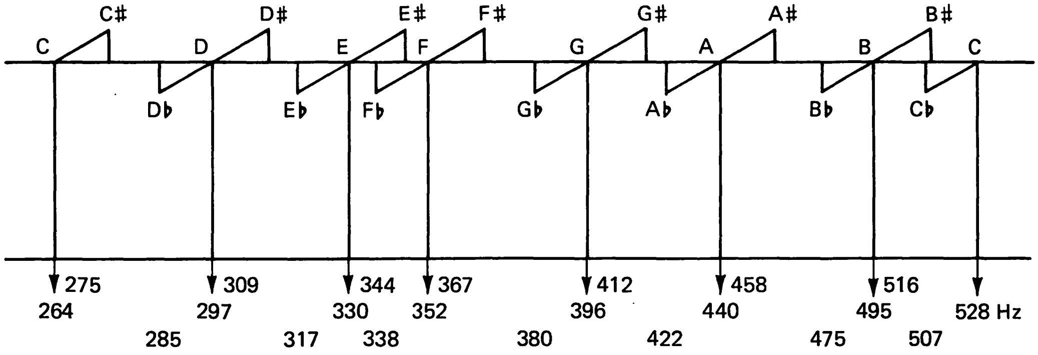


FIGURE 17.5
Frequencies of all notes with their sharps and flats (for fourth octave).

Insert the following constants after MiddleC in the Const statement of Program Music1.

```
D = 2638;              F = 2225;
G = 1978;              A = 1780;
B = 1583;              HighC = 1484;
```

Instead of repeating the main body of Music1 seven times to play each of the above notes, build a procedure that plays all the notes. The plan is to call this procedure one time for each note played. Type in the following, before the main body of your program:

```
Procedure Play (musicnote : Integer);
begin
end;
```

Now "cut and paste" (see your Macintosh manual if you need further explanation) the following statements from the main program to the body of procedure Play:

```
With Note Do
    begin
        Mode                 := -1;          {use square wave synthesizer}
        Triplets.count       := MiddleC;
        Triplets.Duration    := 80;
        Triplets.Amplitude   := 100;
    end;
  StartSound ( @Note, SizeOf(Note), Pointer(-1) );
```

Your procedure should look like procedure Play in Figure 17.6 after pasting the above and changing:

```
Triplets.count := MiddleC;
```

to:

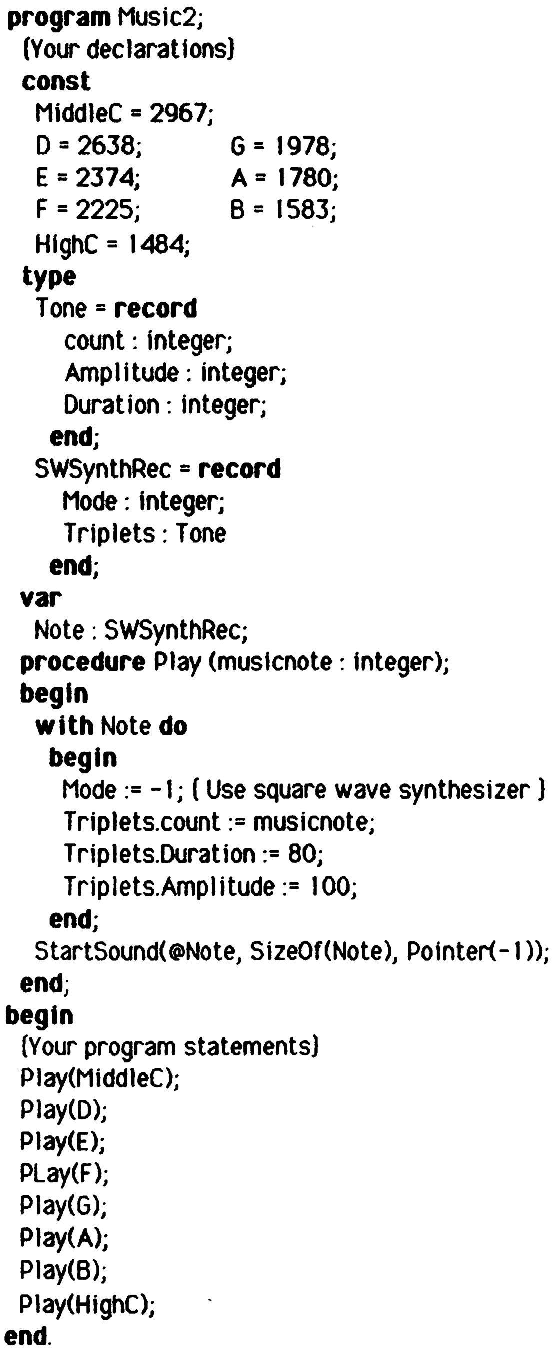
```
Triplets.count := musicnote;
```

Finally, modify Music1 so that procedure Play is called once for each of the eight notes, as shown in Figure 17.6. Save this program as Music2, for later use if you like. Select RUN-GO and listen as your Macintosh plays each note in the octave.

Hands-On "Oh! Susanna"

Not all musical scores are played using the same time duration for all notes. A note can be played for a whole beat, half a beat, and so on. Figure 17.7 (b) lists the encoding of "Oh! Susanna" for the score shown in Figure 17.7 (a).

The sharp sign (#) is used to designate a sharp note. The digit after the note specifies the fraction of the beat that the note should be played. For example, B means to play "B" for one whole beat; A2 means to play "A" ½ of a



```
program Music2;
 {Your declarations}
 const
  MiddleC = 2967;
  D = 2638;          G = 1978;
  E = 2374;          A = 1780;
  F = 2225;          B = 1583;
  HighC = 1484;
 type
  Tone = record
     count : integer;
     Amplitude : integer;
     Duration : integer;
    end;
  SWSynthRec = record
     Mode : integer;
     Triplets : Tone
    end;
 var
  Note : SWSynthRec;
 procedure Play (musicnote : integer);
 begin
  with Note do
   begin
    Mode := -1; { Use square wave synthesizer }
    Triplets.count := musicnote;
    Triplets.Duration := 80;
    Triplets.Amplitude := 100;
   end;
  StartSound(@Note, SizeOf(Note), Pointer(-1));
 end;
begin
 {Your program statements}
 Play(MiddleC);
 Play(D);
 Play(E);
 PLay(F);
 Play(G);
 Play(A);
 Play(B);
 Play(HighC);
end.
```

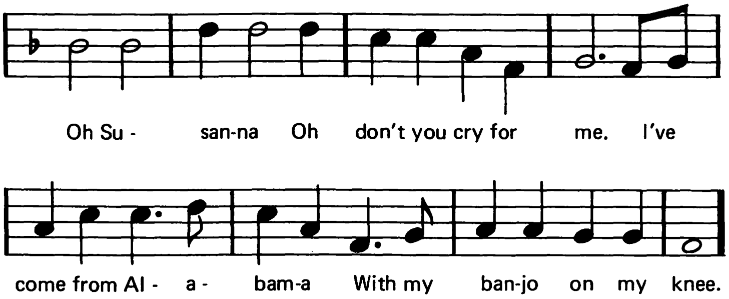
FIGURE 17.6
Listing of Program Music2.

beat; F8 means to play "F" ⅛ of a beat. If you look carefully, some of the notes are followed by a dot (.). The dot means that the length of time the note should be played is one and one-half times the number of its beats. For example, "F." means note F should be played for 1½ whole beats; "A2." means note A should be played for ¾ of a beat.

Figure 17.8 lists a new version of Music2 (called Music3) which has been modified to play "Oh! Susanna." The only changes to Music2 were: (1) changing the Play procedure to take care of variations in the beat of each note, and (2) including a Boolean flag to notify the Play procedure that the dotted notes should be played 50 percent longer.

Hands-On Concerto

Programs Music1, Music2, and Music3 all lack the flexibility needed for playing any general song or melody. Program Concerto shown in Figure 17.9 is (almost) a general program that enables you to play a variety of songs, yet is much easier to use than the previous three programs. When you run Concerto, it prompts



(a) "Oh! Susanna" musical score.

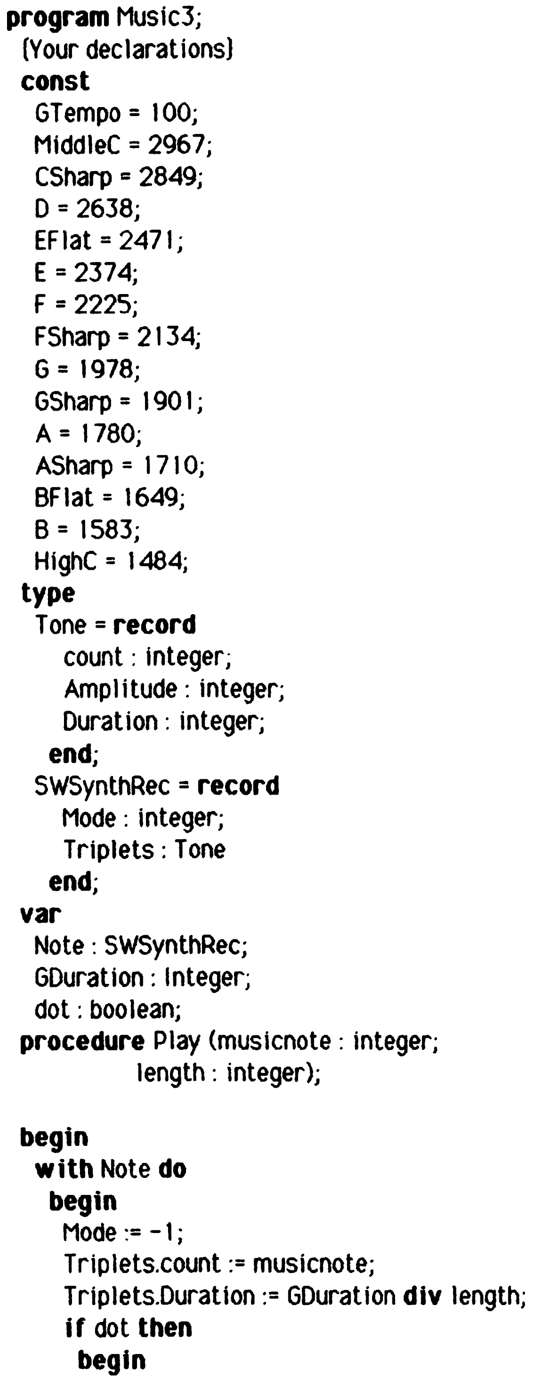
A♯2
A♯2
D4
D2
D4
C4
C4
A4
F4
G2

G.
F8
G8
A4
C4
C4
C.
D8
C4
A4

F4
F.
G8
A4
A4
G4
G4
F2
F.

(b) Encoding of "Oh! Susanna" musical score.

FIGURE 17.7



```
program Music3;
 {Your declarations}
 const
  GTempo = 100;
  MiddleC = 2967;
  CSharp = 2849;
  D = 2638;
  EFlat = 2471;
  E = 2374;
  F = 2225;
  FSharp = 2134;
  G = 1978;
  GSharp = 1901;
  A = 1780;
  ASharp = 1710;
  BFlat = 1649;
  B = 1583;
  HighC = 1484;
 type
  Tone = record
     count : integer;
     Amplitude : integer;
     Duration : integer;
   end;
  SWSynthRec = record
     Mode : integer;
     Triplets : Tone
   end;
 var
  Note : SWSynthRec;
  GDuration : Integer;
  dot : boolean;
 procedure Play (musicnote : integer;
          length : integer);

 begin
  with Note do
   begin
     Mode := -1;
     Triplets.count := musicnote;
     Triplets.Duration := GDuration div length;
     if dot then
      begin
```

FIGURE 17.8
Program Music3.

```
      dot := False;
      Triplets.Duration := Triplets.Duration + round(Triplets.Duration * 0.5);
     end;
    Triplets.Amplitude := 100;
   end;
  StartSound(@Note, SizeOf(Note), Pointer(-1));
 end;
begin
 {Your program statements}
 GDuration := 900 div GTempo * 16; { Duration of a Whole beat }
 dot := False;
 Play(ASharp, 2);
 Play(ASharp, 2);
 Play(D, 4);
 Play(D, 2);
 Play(D, 4);
 Play(MiddleC, 4);
 Play(MiddleC, 4);
 Play(A, 4);
 PLay(F, 4);
 Play(G, 2);
 dot := True;
 Play(G, 1);
 PLay(F, 8);
 Play(G, 8);
 Play(A, 4);
 Play(MiddleC, 4);
 Play(MiddleC, 4);
 dot := True;
 Play(MiddleC, 1);
 Play(D, 8);
 Play(MiddleC, 4);
 Play(A, 4);
 PLay(F, 4);
 dot := True;
 PLay(F, 1);
 Play(G, 8);
 Play(A, 4);
 Play(A, 4);
 Play(G, 4);
 Play(G, 4);
 PLay(F, 2);
 dot := True;
 PLay(F, 1);
end.
```

FIGURE 17.8 *(continued)*

you to enter the notes you want played. After you are done entering your notes, it plays them all. Here is how it works.

Entering Notes

To enter each note, the following format must be used: the NoteName followed by an optional sharp or flat symbol, followed by an optional length, followed by an optional dot.

NoteName/Sharp or Flat /Length /Dot

Here are some examples:

A4	is a note A having a ¼ note duration.
C#16	is a C sharp with a 1/16 note duration.
D2.	is a note D with a ¾ note duration (½ + ¼ for the dot).
B_	is a B flat.

The default octave is always octave 4, # designates a note as sharp, _ is used to designate a note as flat, and . is used to express a dotted note.

When you have finished entering all the notes you want, type either *END* or *end*.

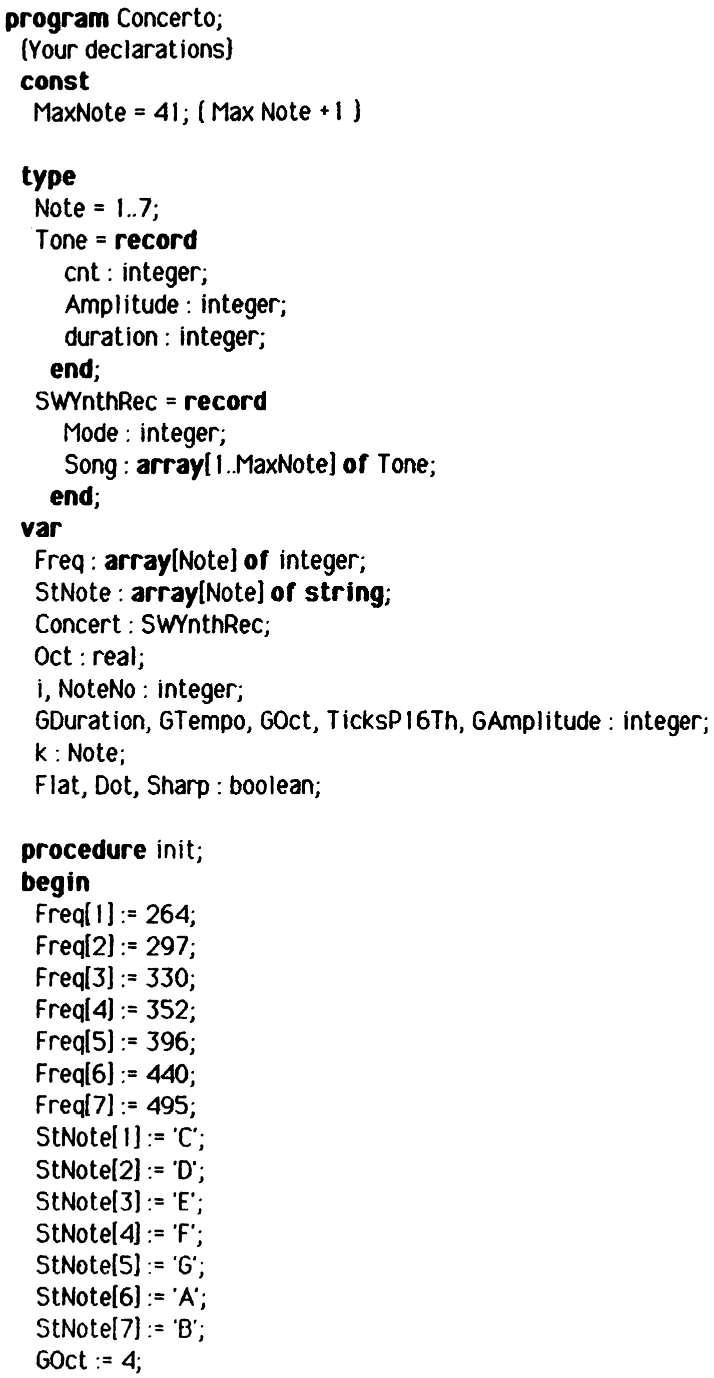
If you'd like to play notes from an octave different from the default fourth octave, use the O command. Typing "O" followed by a number from 1 to 7 changes the default octave to the number selected. The default octave remains at the new selection until you enter another O command. For example:

O3 sets the octave for all notes entered after it, which will be played in the third octave.

Remember, to change back to the default octave (fourth octave) or any other octave you need, enter a new O command.

Assumptions and Limitations of Concerto

Program Concerto assumes that all the symbols entered for the notes are correct, so no error checking is done. Moreover, it does not check for invalid sharps or flats—flats or sharps which have no corresponding black key on the piano keyboard (B#, C_, E_, and F_). It also assumes a default for duration and amplitude. To change these default values modify procedure Init, which handles the initializations. In addition, it assumes the maximum number of notes entered will not exceed 40. To change this limit, modify MaxNote in the Const statement.



```
program Concerto;
 {Your declarations}
 const
  MaxNote = 41; { Max Note +1 }

 type
  Note = 1..7;
  Tone = record
     cnt : integer;
     Amplitude : integer;
     duration : integer;
    end;
  SWYnthRec = record
     Mode : integer;
     Song : array[1..MaxNote] of Tone;
    end;
 var
  Freq : array[Note] of integer;
  StNote : array[Note] of string;
  Concert : SWYnthRec;
  Oct : real;
  i, NoteNo : integer;
  GDuration, GTempo, GOct, TicksP16Th, GAmplitude : integer;
  k : Note;
  Flat, Dot, Sharp : boolean;

 procedure init;
 begin
  Freq[1] := 264;
  Freq[2] := 297;
  Freq[3] := 330;
  Freq[4] := 352;
  Freq[5] := 396;
  Freq[6] := 440;
  Freq[7] := 495;
  StNote[1] := 'C';
  StNote[2] := 'D';
  StNote[3] := 'E';
  StNote[4] := 'F';
  StNote[5] := 'G';
  StNote[6] := 'A';
  StNote[7] := 'B';
  GOct := 4;
```

***FIGURE 17.9** (continued)*

```
 GAmplitude := 100;
 GTempo := 75;
 TicksP16TH := 900 div GTempo;
 Gduration := TicksP16TH * 16; { Whole Duration}
end;
function Octave (N : note;
         OctNum : integer) : real;
 var
  O, m : integer;
  Temp : real;
begin

 if OctNum = 4 then
  Octave := Freq[N]
 else if OctNum < 4 then
  begin
   O := 4 - OctNum;
   Temp := Freq[N];
   for m := 1 to O do
    Temp := Temp / 2;
   Octave := Temp;
  end
 else
  begin
   O := OctNum - 4;
   Temp := Freq[N];
   for m := 1 to O do
    Temp := Temp * 2;
   Octave := Temp;
  end;
end;
function Convert (str : string) : integer;
 var
  i, len, No : integer;
  str1 : char;
begin
 len := length(str);
 No := 0;
 for i := 1 to len do
  begin
   str1 := copy(str, i, 1);
   No := ord(str1) - ord('0') + No * 10;
  end;
```

FIGURE 17.9 (continued)

```
  Convert := No;
 end; { Convert }
 function FindNote (str : string) : Note;
  var
   index : Note;
   found : boolean;
 begin
{ Note: we are assuming the note exist for sure }
  found := false;
  index := 1;
  while not found do
   begin
    if StNote[index] = str then
     begin
      FindNote := index;
      found := true;
     end
    else
     index := Succ(index);
   end;
 end;{ FindFreq}
 function Count (Frequency : real) : integer;
 begin
  Count := round(783360 / Frequency);
 end;
 procedure FndDotShFl (var str : string);
  var
   str1 : string;
 begin
  str1 := copy(str, 1, 1);
  Sharp := false;
  Dot := false;
  Flat := false;
  if str1 = '*' then
   begin
    Sharp := true;
    delete(str, 1, 1);
   end
  else if str1 = '_' then
   begin
    Flat := true;
    delete(str, 1, 1);
   end;
```

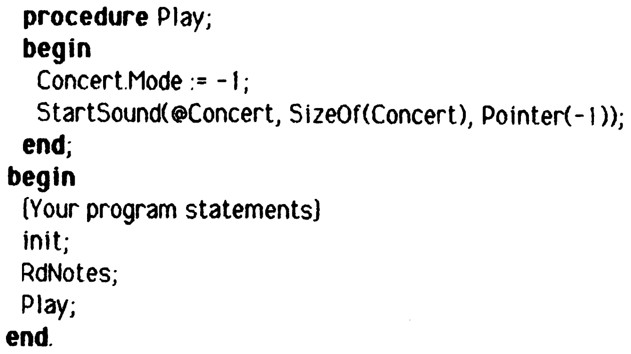
FIGURE 17.9 (continued)

```
  str1 := copy(str, length(str), 1);
  if str1 = '.' then
    begin
      delete(str, length(str), 1);
      Dot := true;
    end;
end;
procedure BuildNote (str : string);
  var
    str1, str2 : string;
    temp, period : integer;
    frq : real;
    index : Note;
begin
  str1 := copy(str, 1, 1);
  index := FindNote(str1);
  period := 1;
  if length(str) = 1 then
    begin
      with Concert do
        begin
          Song[NoteNo].Duration := GDuration;
          Song[NoteNo].Amplitude := GAmplitude;
          Song[NoteNo].cnt := Count(Octave(index, GOct));
        end
    end
  else
    begin
      delete(str, 1, 1);
      FndDotShFl(str);
      if length(str) > 0 then
        period := Convert(str);
      frq := Octave(index, Goct);
      if Flat then
        frq := frq * 24 / 25
      else if Sharp then
        frq := frq * 25 / 24;
      Temp := Gduration div period;
      if Dot then
        Temp := Round(Temp * 1.5);
      with Concert do
        begin
          Song[NoteNo].Duration := Temp;
          Song[NoteNo].Amplitude := GAmplitude;
```

***FIGURE 17.9** (continued)*

```
     Song[NoteNo].cnt := Count(frq);
    end
  end
end; { of Build Note }
procedure RdNotes;
 var
  done : boolean;
  str, str1 : string;
begin
 writeln('To End Entering the Notes Type End for the Note');
 Writeln;
 writeln('Enter the Notes :  ');
 done := False;
 NoteNo := 1;
 repeat
  readln(str);
  if (str = 'end') or (str = 'END') then
   begin
    done := true;
    with Concert do
     begin
      Song[NoteNo].Duration := GDuration;
      Song[NoteNo].Amplitude := GAmplitude;
      Song[NoteNo].cnt := 0;
     end
   end
  else
   begin
    str1 := Copy(str, 1, 1);
    if str1 = '0' then
     begin
      str1 := Copy(str, 2, 1);
      GOct := Convert(str1);
     end
    else
     begin
      BuildNote(str);
      NoteNo := NoteNo + 1
     end
   end
 until done;
end;
```

***FIGURE 17.9** (continued)*



```
 procedure Play;
 begin
  Concert.Mode := -1;
  StartSound(@Concert, SizeOf(Concert), Pointer(-1));
 end;
begin
 {Your program statements}
 init;
 RdNotes;
 Play;
end.
```

***FIGURE 17.9** (continued)*

Structure of Concerto

Program Concerto is composed of three modules: initialization, reading notes, and playing notes. The following is the main body of Concerto, which calls a procedure for doing each of these three actions.

```
Begin
    Init;
    RdNotes;
    Play
End.
```

Initialization: Procedure Init stores the frequencies of the default octave in array *Freq* and sets the values of *Gduration, GOct* (for the default octave), and *GTemp* (for the default amplitude).

Reading Notes: Procedure RdNotes reads the entered notes and calls *BuildNote* to extract the musical information from the input string.

Procedure BuildNote uses *FndDotShFl, Octave, FindNote,* and *Convert* to determine the frequency and length of each entered note. It then stores this information in the *Song* array (see the definition of Concert in the listing). Function *FindNote* returns the type of note entered (C,D,E,F,G,A,B). To determine whether a note is sharp, flat, and/or a dotted note, BuildNote calls the *FndDotShFl* procedure. Procedure *Octave* is used to get the frequency of the octave; function *Convert* is a peripheral function that returns a numerical value for the duration of a note (if its length has been specified).

Summary

What you've been exposed to so far is only one of three different methods for generating sound using the Macintosh. As a matter of fact, you learned the simplest method of sound generation, which uses the square-wave synthesizer. Using the *four-tone synthesizer* you can generate tones with up to four voices producing sound simultaneously. Using the *free-form synthesizer,* you can create complex music and speech. These other two methods require much more knowledge of the physics of sound and music than we have discussed here.

Problem Solving

1. Normal, staccato, and legato are different musical styles. Normal notes have enough pause between them to make then distinct. Staccato notes have short breaks between them, and legato notes have no break between them.

   Add an S command to Concerto to provide a staccato style by holding each note for ¾ of its length, an L command to provide a legato style that plays each note to its full length, and an N to provide a normal style by playing each note ⅞ of its length with a pause or rest for the remaining ⅛ of its length.
2. Add new command LNnn to set the length of the notes globally. All notes entered after the LNnn command will be played according to the value of nn, where nn is an integer from 1 to 64. Here are some examples:

nn = 1	for a whole note
nn = 2	for a ½ note
nn = 8	for a ⅛ note
nn = 64	for a 1/64 note

3. Modify Concerto so that it will not accept the notes that do not have a corresponding black key on the piano.
4. Modify Concerto to save the notes on the disk for later use.
5. Write a program that reads the notes of a melody from disk and then plays them.

Session 18:

Graphics

In this session you will learn the elements of graphics using the Macintosh Quickdraw procedures for drawing rectangles, polygons, and icons. We will introduce you to pixels and other new terms used to understand how the Macintosh draws.

The Drawing Window

If the drawing Window is not already visible on the Macintosh screen, select it from the Windows menu. This is the area where all graphics output will appear. The Drawing Window is a window into a large two-dimensional grid which contains picture elements called *pixels.* Each pixel is a small dot which can be either black or white depending on its setting. We often use 1 to indicate a black pixel setting and 0 to indicate a white pixel setting. Since pixels can only be 0 or 1 in value, we also call them *bits.*

A picture is merely a mosaic of black and white pixels. Macintosh Pascal uses a collection of intrinsic functions and procedures called the *Quickdraw library* to perform the necessary details for setting pixels. In this session, you will use only a few of the Quickdraw routines to draw several simple pictures in the Drawing Window.

The routines are in two separate libraries called Quickdraw1 and Quickdraw2, respectively. In the advanced hands-on exercises shown later in this session, you will need a *uses statement* at the top of your program. The uses statement tells Pascal which library routines to use during the execution of a program.

Two-Dimensional Grid

The pixels in the Drawing Window are located by giving two numbers: a horizontal coordinate and a vertical coordinate. Figure 18.1 shows how each number ranges from -32,768 to +32,767. The upper left-hand corner of the Drawing Window is assumed to be at location (0, 0).

To get a quick idea of how a picture is drawn from a running Pascal program, enter and run the simple program shown in Figure 18.2. This program moves an imaginary pen around the two-dimensional grid and "draws" the

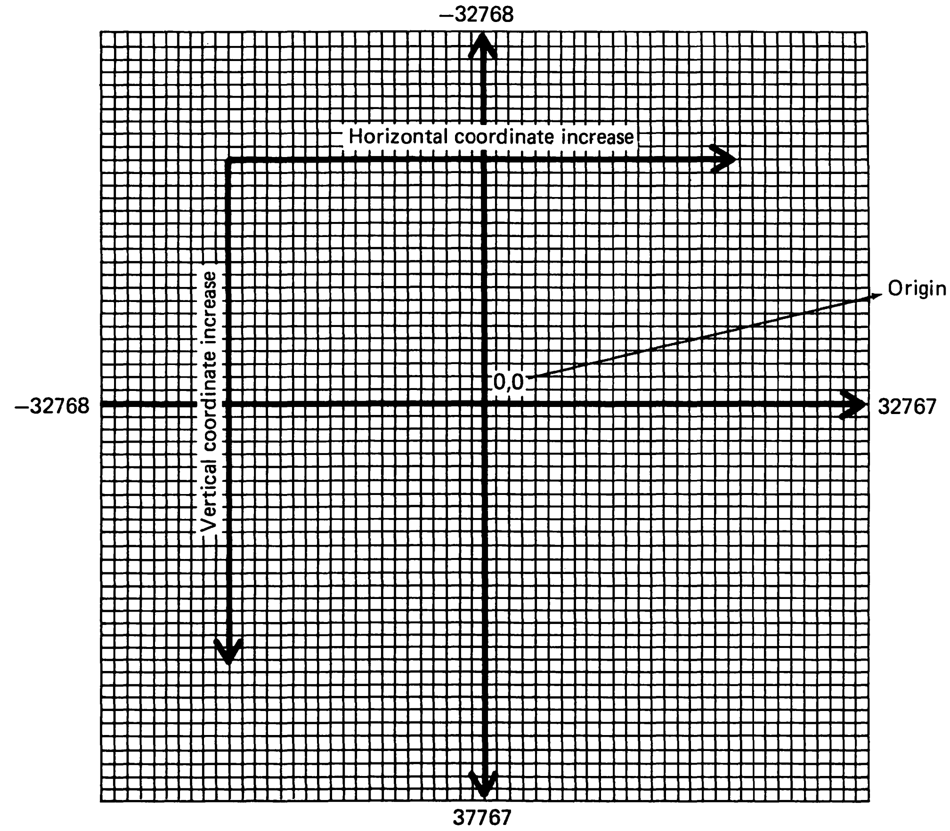
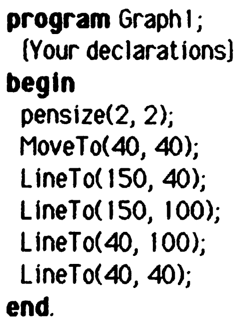


FIGURE 18.1
Horizontal and vertical coordinates increase direction.



```
program Graph1;
  {Your declarations}
begin
  pensize(2, 2);
  MoveTo(40, 40);
  LineTo(150, 40);
  LineTo(150, 100);
  LineTo(40, 100);
  LineTo(40, 40);
end.
```

FIGURE 18.2
Hands-on drawing simple lines.

figures you command it to draw. Program Graph1 selects the pen size (width of lines to be drawn) and then draws a rectangle with corners at pixels (40, 40), (150, 40), (150, 100), and (40, 100). Let's study how program Graph1 does its work in greater detail.

```
Pensize(2,2);
```

Pensize sets the width of the drawing styles. Consult the *Macintosh Pascal Reference Manual* for a list of different drawing sizes.

```
Move(40,40);
```

Move moves the pen to a point with horizontal and vertical coordinates (40, 40), respectively.

```
LineTo(150,40);
```

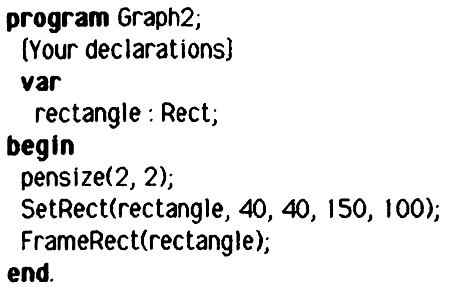
This procedure draws a straight line from the previous pixel at (40, 40) to a pixel with coordinates equal to (150, 40).

The other LineTo procedure calls complete the picture by drawing one side of the rectangle each.

Rectangles

Instead of drawing the four sides of each rectangle every time you want a rectangle in a picture, you can use other intrinsic functions that draw entire rectangles in one operation. Automatic rectangles can be defined by specifying the left-top and right-bottom corner coordinates. This short cut method of drawing rectangles is illustrated in Figure 18.3.

Enter Program Graph2 (Figure 18.3) and select RUN-GO. Notice that Program Graph2 produces the same result as Program Graph1, but Graph2 uses objects of predefined type Rect.



```
program Graph2;
 (Your declarations)
 var
  rectangle : Rect;
begin
 pensize(2, 2);
 SetRect(rectangle, 40, 40, 150, 100);
 FrameRect(rectangle);
end.
```

FIGURE 18.3
Rectangles.

```
Type VHselect = (V,H)
   Point = Record
            Case Integer Of
                 0 : ( V, H : Integer )
                 1 : (Vh : Array [VHselect] of Integer);
            end;
   Rect  = Record
            Case Integer Of
                 0 : (top, bottom, right, left : Integer);
                 1 : (topleft, bottright : Point);
            end;
            Var
                 Rectangle : Rect;
```

Variable Rectangle holds the necessary information for drawing a rectangle whose two corner coordinates are given by procedure SetRect.

```
SetRect (Rectangle, 40, 40, 150, 100 );
```

Calling Procedure SetRect is equivalent to executing the following assignment statements:

```
With Rectangle Do
      begin
               top    := 40;
               left   := 40;
               bottom := 150;
               right  := 100;
      end;
```

In general, the parameters of SetRect are:

```
SetRect (Rectangle, top, left, bottom, right);
```

Next, your program must use the information set up by SetRect to actually draw the sides of a rectangle. One way to do this is to draw a square frame using Procedure FrameRect.

```
FrameRect (Rectangle);
```

The width and height of the sides are specified by the width and height of the pen which is set in Procedure Pensize. In this example, both the width and height of Pen are set to 2.

```
PenSize (2,2);
```

Experiment with different line widths and heights by changing the parameters of PenSize to (1, 1), (1, 2), or (1, 3) and select RUN-Go for each case.

Operations on Rectangles

Many operations can be done on objects of type Rectangle. For example, rectangles can be moved around by adding or subtracting an offset value to each of their coordinates. The contents of the rectangular areas can be erased or filled with different patterns.

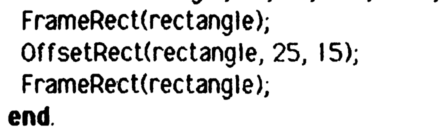
Add the following procedure calls before the End statement of Graph2 and select RUN-GO.

```
OffsetRect (Rectangle, 25, 15);
FrameRect (Rectangle);
```

Figure 18.4 shows the program after this change. As you can see, a new rectangle is framed, but offset from the original by (25, 15). Now change (25, 15) to (-25, -15) and select RUN-GO again.

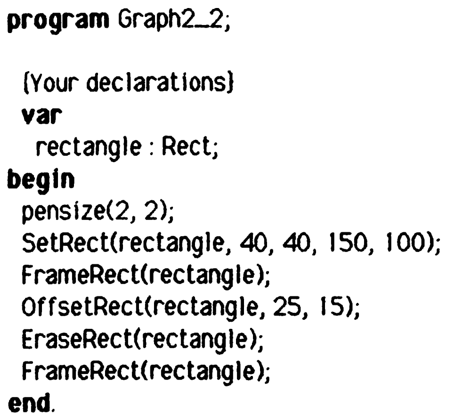
Generally, if the offset values are positive, the movement is to the right or down; if they are negative, the movement will be the reverse, left or up.

In some cases, you may want to clear or erase the area within a rectangle. To do so, call Procedure EraseRect. Figure 18.5 shows how to erase the area within the second rectangle created by offsetting the first one before framing it.



```
program Graph2_1;

 {Your declarations}
 var
  rectangle : Rect;
begin
 pensize(2, 2);
 SetRect(rectangle, 40, 40, 150, 100);
 FrameRect(rectangle);
 OffsetRect(rectangle, 25, 15);
 FrameRect(rectangle);
end.
```

FIGURE 18.4
Moving rectangles.



```
program Graph2_2;

 {Your declarations}
 var
  rectangle : Rect;
begin
 pensize(2, 2);
 SetRect(rectangle, 40, 40, 150, 100);
 FrameRect(rectangle);
 OffsetRect(rectangle, 25, 15);
 EraseRect(rectangle);
 FrameRect(rectangle);
end.
```

FIGURE 18.5
More on rectangles.

Rounded-Corner Rectangles

There is a procedure to draw rectangles with rounded corners. In addition to the coordinates of the top-left and bottom-right corners, you must provide the oval width and height (see Figure 18.6).

Modify your program to incorporate these changes and select RUN-GO to see their effect. The oval height and oval width of the corners are 30 and 15, respectively.

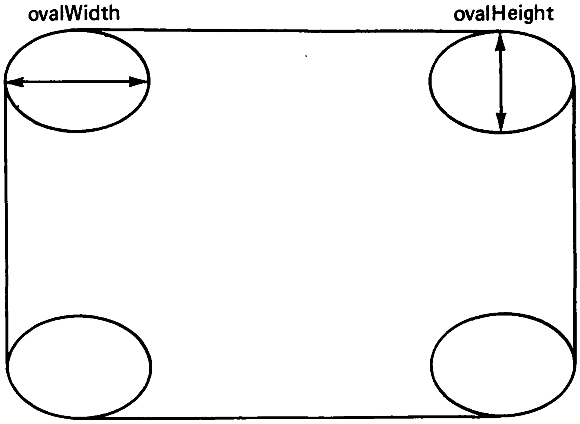
Now change (30, 15) to (15, 30), (10, 5), and (10, 10). Select RUN-GO in each case to see the different results.

Regions

Any arbitrary area or set of areas in the two-dimensional grid is called a *region.* A rectangle can be considered the simplest form of a region. You create a region each time you draw lines or shapes such as rectangles and ovals.

One of the most significant features of the Macintosh is its ability to capture and store information concerning a region and then "play back" the region at a later time. To capture the definition of a region your program must have a variable of predefined type Region to hold the graphical information of the region.

```
Type Region = Record
              Rgnsize   : Integer;
              RgnBbox : Rect;
              {optional region definition data . . }
              end;
```



(a) Rounded-Corner Rectangle

```
program Graph2_4;
(Your declarations)
var
  rectangle:Rect;
begin
  pensize(2, 2);
  SetRect(rectangle, 40, 40, 150, 100);
  FrameRoundRect(rectangle, 30, 15);
  OffsetRect(rectangle, 25, 20);
  EraseRect(rectangle);
  FrameRect(rectangle);
end

{draw it}
```

(b) Rounded-Corner Program Listing

FIGURE 18.6

Rgnsize contains the size of the region in bytes; RgnBbox is a rectangle which completely encloses the region; the remaining components information used to draw the picture.

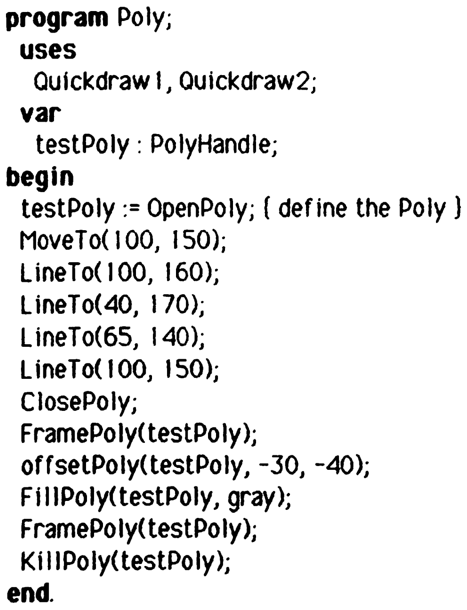
Two of the most useful regions are called *Pictures* and *Polygons.* In the following sections, you will use regions to draw and play back pictures and polygons.

Polygons and Pictures

Polygons and Pictures are graphical objects which hold pen movements to be played back later, using a simple procedure call. Both Polygons and Pictures are objects containing a sequence of connected lines which are manipulated as a single region. Pictures can be scaled to different sizes, whereas the size of a polygon cannot be changed.

Hands-On Polygons

Program Poly in Figure 18.7 is an example of building a polygon region in memory and replaying it later. Libraries Quickdraw1 and Quickdraw2 are needed because they contain the procedures for processing regions. Enter Program Poly and select RUN-GO.



```
program Poly;
 uses
  Quickdraw1, Quickdraw2;
 var
  testPoly : PolyHandle;
begin
 testPoly := OpenPoly; { define the Poly }
 MoveTo(100, 150);
 LineTo(100, 160);
 LineTo(40, 170);
 LineTo(65, 140);
 LineTo(100, 150);
 ClosePoly;
 FramePoly(testPoly);
 offsetPoly(testPoly, -30, -40);
 FillPoly(testPoly, gray);
 FramePoly(testPoly);
 KillPoly(testPoly);
end.
```

FIGURE 18.7
Hands-on creating polygons.

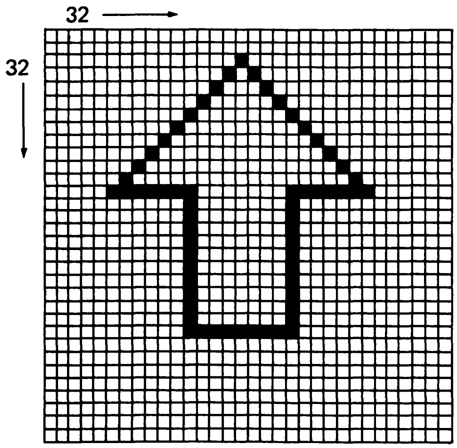


FIGURE 18.8
32 x 32 rectangle for defining an icon.

OpenPoly opens an object called testPoly, which captures and saves all subsequent pen movements. MoveTo and LineTo are written to testPoly until the ClosePoly procedure is executed; then FramePoly draws the polygon stored in testPoly. You can replay the polygon stored in testPoly as many times as you want in any form. For example:

```
OffSetPoly (testPoly, -30, -40);     {move it}
FillPoly (testPoly, gray);           {fill it with gray}
FramePoly (testPoly);                {draw it{
```

These operations create another Polygon offset by (-30, -40) and fill it with a gray background.

When you're done with a polygon, you should destroy it so that the storage can be used by Quickdraw for other purposes.

```
KillPoly (testPoly);          {release storage}
```

Creating Icons

An *icon* is a 32 × 32 pixel pattern. Figure 18.8 shows a 32 × 32 rectangle containing an icon. To create an icon, you set each of the 32 × 32 pixels to either 1 or 0. A 1 corresponds to a black pixel and a 0 to a white pixel.

A more compact way to enter the 32 × 32 pixel pattern is to use hexadecimal numbers; one hexadecimal number equals four bits. For example, the first five rows of Figure 18.8 are shown in binary and hexadecimal as follows.

1. Binary form

```
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0000 0000 0000 0000 0000
0000 0000 0000 0001 0000 0000 0000 0000
0000 0000 0000 0010 1000 0000 0000 0000
0000 0000 0000 0100 0100 0000 0000 0000
---- ---- ---- ---- ---- ---- ---- ----
 |    |    |    |    |    |    |    |
 0    0    0    4    4    0    0    0
```

2. Hexadecimal form

```
00000000
00000000
00010000
00028000
00044000
```

Use the following table for conversion.

Binary		Hexadecimal
0000	=	0
0001	=	1
0010	=	2
0011	=	3
0100	=	4
0101	=	5
0110	=	6
0111	=	7
1000	=	8
1001	=	9
1010	=	A
1011	=	B
1100	=	C
1101	=	D
1110	=	E
1111	=	F

Enter program Icons (Figure 18.9) and select RUN-GO. This program uses ManFace of type *IconData* to store the pattern. IconData is an array of 32 elements of type *LongInt*, and since each LongInt consists of 32 bits, each element of ManFace can store one row of the icon pattern.

The intrinsic procedure for drawing icons, *PlotIcon*, uses a pointer to a location in memory which contains the address of the place you've stored your icon pattern. A pointer to an address is called a *handle*. The following type definitions are needed to declare handles and pointers to locations in memory where icon patterns are stored.

```
Type
    IconData   =Array[0 .. 31] of LongInt;     {the pattern}
    IconPtr    =^IconData;                     {address of pattern}
    IconHandle=^IconPtr;                       {handle to pattern}
```

From the type declarations above, you can define the following variables:

```
    ManFace    :IconData;             {to keep actual pattern}
    FacePtr    :IconPtr;              {a pointer to pattern}
    FaceHandle:IconHandle;            {pointer to the pointer pattern}
    Where      :Rect;                 {rectangle that icon is drawn within}
```

StuffHex converts a string of characters into an equivalent group of hexadecimal numbers and then stores the binary equivalent of the hexadecimal numbers in a specific location in memory. In Figure 18.9 each StuffHex procedure call converts a string of hex characters and then copies the hex values to four elements of array ManFace. After the first call to StuffHex, the first four elements of ManFace contain the following.

ManFace [0]is01FFFE00
ManFace [1]is03000300
ManFace [2]is06000180
ManFace [3]is0C0000C0

The @ operator extracts the address of an object—@ is the inverse of °. For example, to obtain the address of array ManFace make the following assignment:

FactPtr := @FacePtr;

This is read as "FacePtr is assigned the address of ManFace."

Similarly, the address of variable FacePtr can be obtained by assigning the "address of FacePtr" to FaceHandle.

FaceHandle := @FacePtr;

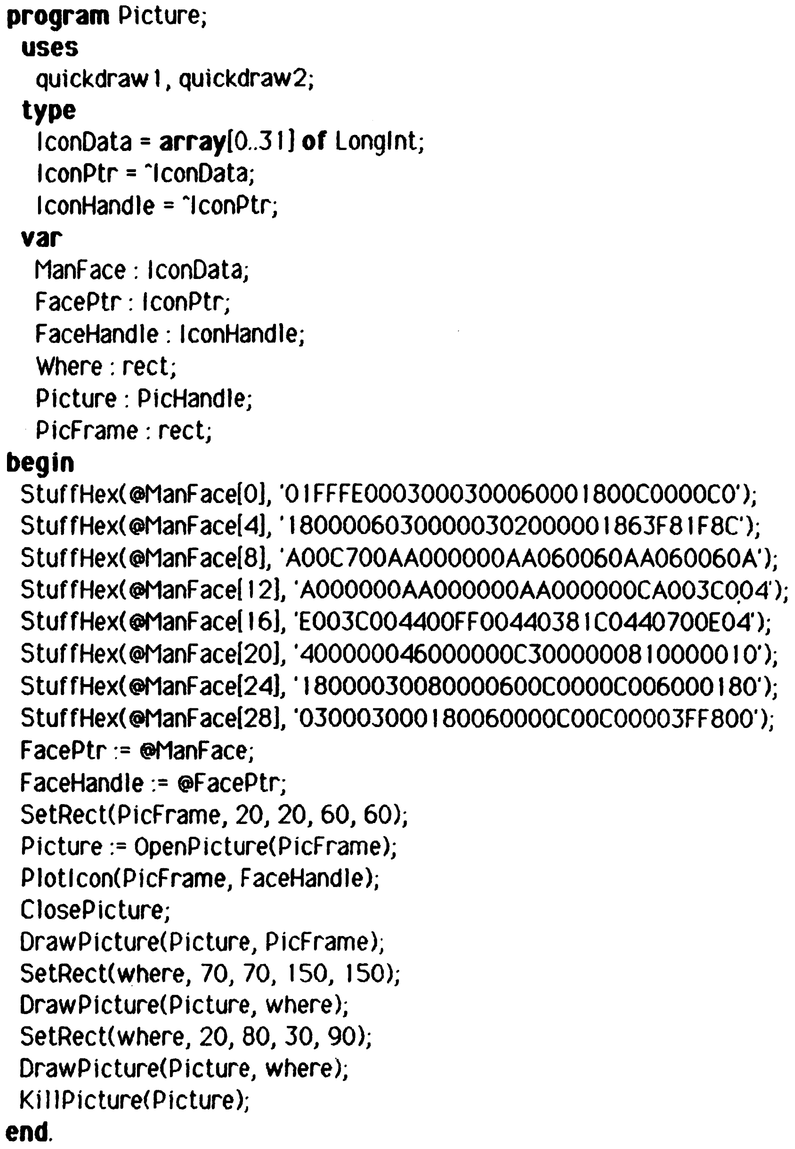
```
program Icons;
 uses
  quickdraw1, quickdraw2;
 type
  IconData = array[0..31] of LongInt;
  IconPtr = ^IconData;
  IconHandle = ^IconPtr;
 var
  ManFace : IconData;
  FacePtr : IconPtr;
  FaceHandle : IconHandle;
  Where : rect;
begin
 StuffHex(@ManFace[0], '01FFFE00030003000600001800C0000C0');
 StuffHex(@ManFace[4], '18000060300000302000001863F81F8C');
 StuffHex(@ManFace[8], 'A00C700AA000000AA060060AA060060A');
 StuffHex(@ManFace[12], 'A000000AA000000AA000000CA003C004');
 StuffHex(@ManFace[16],.'E003C004400FF00440381C0440700E04');
 StuffHex(@ManFace[20], '400000046000000C3000000810000010');
 StuffHex(@ManFace[24],.'18000030080000600C0000C006000180');
 StuffHex(@ManFace[28], '030003000180060000C00C00003FF800');
 FacePtr := @ManFace;
 FaceHandle := @FacePtr;
 SetRect(where, 20, 20, 60, 60);
 PlotIcon(Where, FaceHandle);
end.
```

FIGURE 18.9

Program Icons.

The rest of Program Icons is easy. First, define the rectangular area where you want to draw an icon and simply call PlotIcon. PlotIcon takes the pattern stored in FaceHandle and copies it to the rectangle called Where. The Quickdraw system displays the rectangular area on the screen at the location given by SetRect.

```
SetRect (Where, 20, 20, 60, 60);
PlotIcon (Where, FaceHandle);
```



```
program Picture;
 uses
  quickdraw1, quickdraw2;
 type
  IconData = array[0..31] of LongInt;
  IconPtr = ^IconData;
  IconHandle = ^IconPtr;
 var
  ManFace : IconData;
  FacePtr : IconPtr;
  FaceHandle : IconHandle;
  Where : rect;
  Picture : PicHandle;
  PicFrame : rect;
begin
 StuffHex(@ManFace[0], '01FFFE000300030006000 1800C0000C0');
 StuffHex(@ManFace[4], '1800006030000030200000 1863F81F8C');
 StuffHex(@ManFace[8], 'A00C700AA000000AA060060AA060060A');
 StuffHex(@ManFace[12], 'A000000AA000000AA000000CA003C004');
 StuffHex(@ManFace[16], 'E003C004400FF0044038 1C0440700E04');
 StuffHex(@ManFace[20], '400000046000000C30000008 10000010');
 StuffHex(@ManFace[24], '18000030080000600C0000C006000180');
 StuffHex(@ManFace[28], '030003000 180060000C00C00003FF800');
 FacePtr := @ManFace;
 FaceHandle := @FacePtr;
 SetRect(PicFrame, 20, 20, 60, 60);
 Picture := OpenPicture(PicFrame);
 PlotIcon(PicFrame, FaceHandle);
 ClosePicture;
 DrawPicture(Picture, PicFrame);
 SetRect(where, 70, 70, 150, 150);
 DrawPicture(Picture, where);
 SetRect(where, 20, 80, 30, 90);
 DrawPicture(Picture, where);
 KillPicture(Picture);
end.
```

FIGURE 18.10
Program Pictures.

Hands-On Pictures

Pictures are in many respects similar to polygons except pictures can be scaled—changed to a different size. In this hands-on experiment you will modify the icon program of Figure 18.9 to capture a picture, move it, and then display it in a reduced size. Change Program Icons to get a program called Pictures (Figure 18.10). Select RUN-GO to see the results shown in Figure 18.11.

The middle-sized face shown in the upper-left corner is the original icon drawn in Rectangle (20, 20, 60, 60). The enlarged version is drawn in the larger Rectangle (70, 70, 150, 150); the reduced version is drawn in the smaller Rectangle (70, 80, 30, 90).

To set up and use a picture, first define a handle and a region.

```
Var
  Picture   : PicHandles;      {pointer to pointer}
  PicFrame  : Rect;            {region}
```

Next set the initial picture size and open a region for it.

```
SetRect (PicFrame, 20, 20, 60, 60);
Picture := OpenPicture (PicFrame);
```

Now, all operations on Picture are "remembered" so they can be played back whenever you want. In Figure 18.10 you only need to remember one operation—the drawing of ManFace.

```
PlotIcon (PicFrame, FaceHandle);
```

To prevent other operations from being recorded as well, you must close the picture frame.

```
ClosePicture;
```

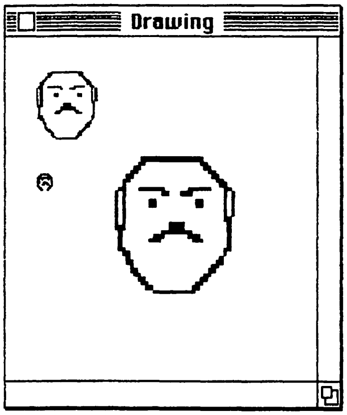


FIGURE 18.11
Scaling pictures.

From here on, you can use Picture to draw different-sized replicas of the original picture.

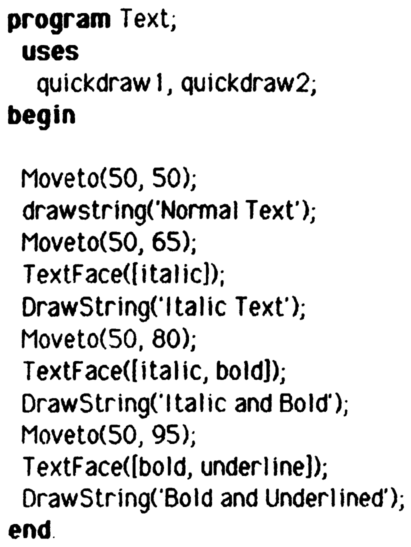
```
DrawPicture (Picture, PicFrame);        {mamma bear: mid-size}
SetRect (Where, 70, 70, 150, 150);      {papa bear: large-size}
DrawPicture (Picture, Where);
SetRect (Where, 20, 80, 30, 90);        {baby bear: small-size}
DrawPicture (Picture, Where);
```

When you are done, destroy the object which contains your picture.

```
KillPicture (Picture);          {release object}
```

Hands-On Text Drawings

The Quickdraw library provides a set of procedures for defining the shape and size of textual information to be displayed. For example, you may decide to draw characters as bold, bold and underlined, and so on. This is done by calling the appropriate procedure to set up the characteristics of the textual data before drawing. Program Text (Figure 18.12) is an example of how to do this. Enter it and select RUN-GO to see how Quickdraw handles drawing textual information.



```
program Text;
 uses
  quickdraw1, quickdraw2;
begin

 Moveto(50, 50);
 drawstring('Normal Text');
 Moveto(50, 65);
 TextFace([italic]);
 DrawString('Italic Text');
 Moveto(50, 80);
 TextFace([italic, bold]);
 DrawString('Italic and Bold');
 Moveto(50, 95);
 TextFace([bold, underline]);
 DrawString('Bold and Underlined');
end.
```

FIGURE 18.12
Program Text.

Drawing Environment: GrafPort

Imagine yourself in an art supply shop where you have access to many drawing boards simultaneously and you are able to draw on each of them any time you want. Furthermore, assume you have a separate set of brushes for each board and you can adjust the size and background patterns of the canvas on each drawing board. Since all drawing boards are equipped with their own set of tools, none of them affects any of the others. Quickdraw provides a similar facility which enables you to define different drawing boards called *GrafPorts.*

You can have more than one GrafPort in your program at the same time, each with its own characteristics (background patterns, pen or brush size, etc.). GrafPorts are the basis of multiple windows in Macintosh; each window is associated with one GrafPort. Refer to your *Macintosh Pascal Reference Manual* for more details on GrafPorts.

Summary

Quickdraw is the graphics library through which you can define points, lines, regions, rectangles, ovals, polygons, and pictures. When using regions, you must declare variables of certain predefined types before creating objects of these types.

Quickdraw assumes all the drawings occur in a two-dimensional grid. The horizontal coordinates in the grid increase from left to right, and vertical coordinates increase from top to bottom.

Problem Solving

1. Create your own icon by hand and use Program Icons (Figure 18.9) to draw your icon.
2. Write a program to draw a 32 × 32 pixel grid and accept input from the mouse for defining an icon.
3. Write a procedure to print out the hex values for the icon created by the program in Problem 1.
4. Write a program to track the mouse and draw a "rubber-band" line from the point where the mouse button is pressed to the point where the button is released.
5. Write a graphics program to display any card from a deck of playing cards.

Appendix A:

An Overview of Pascal

Pascal programs consist of data and instructions. The data reside in a section called the *data declaration part,* and the instructions reside in a section called the *program body.* All programs begin with a program header (the first line containing the program's name) and end with a period.

```
Program MAIN;
      { Data Declaration Part}
Begin
      { Program Body }
End.
```

Data Declaration Part

The *data declaration part* has many optional subsections. The simplest form contains a list of variables along with the type of each variable. Recall that a type is a set of values. For example, all of the integer values between -32,768 and +32,767 make up a set called *integer* values in Pascal.

Another set of values in Pascal is called *char* because it contains all of the single-letter characters that Pascal programs can process. Thus, 'a', 'b', . . . 'z' and 'A', 'B', . . . 'Z' (letter characters) and the remaining keyboard characters make up the set of values called char. Character values are enclosed in single quotes to distinguish them from names.

A simple data declaration part contains a list of variables and their types. The *reserved word* (a word having special meaning in Pascal) *Var* is used to specify all variables in a program.

```
Var
    ALFA : Char;
    PAY  : Real;
    AGE  : Integer;
    YEAR : Integer;
```

This list can be abbreviated using commas whenever two or more variables belong to the same type.

```
Var
    ALFA       : Char;
    PAY        : Real;
    AGE, YEAR  : Integer;
```

The data declaration part may optionally contain a list of constants. A *constant* is an object much like a variable, except that a constant can never be changed.

```
Const
  N = 10;                 { Integer }
  A = 'M'                 { Char }
  X = 5.2                 { Real }
```

The value of each constant is associated with the name of each constant. The type is inferred from the constant value. We've noted the type in each case by enclosing it between curly brackets. Optional notes like these are called *comments* and have no effect on the program; their purpose is solely to make the program easier to read and understand.

N is an integer constant equal to 10; A is a character constant equal to "M"; and X is a real constant equal to 5.2.

Pascal programmers can invent new types by declaring them in an optional type statement in the data declaration part. A new type is simply a combination of other (previously defined) types.

```
  Type
      DAY    = Integer;
      YEAR   = Integer;
      MONTH  = 1 .. 12;                  { subrange of Integer }
      DATE   = Record
                 D  : DAY;
                 YR : YEAR;
                 MO : MONTH;
                End;
Var
      X,Y,Z      : DATE;                 { user defined type}
```

In the example above, DAY, YEAR, MONTH, and DATE are the names of new types created by the programmer. MONTH is a restricted integer because it can take on the values 1 through 12 and no others. DATE is a new type with three components; D, YR, and MO. A Pascal *record* is a data type consisting of components—each component possibly of different types.

X, Y, and Z are the names of three variables, each of type DATE. To access a component of a DATE variable, you must use a dot notation as shown below.

```
X.D                   { D component of X }
X.YR                  { YR component of X }
X.MO                  { MO component of X }
```

Finally, the data declaration part contains optional *subprograms* defined for a program. A subprogram is either a function or a procedure. In both cases, the subprogram consists of its own data and instructions. The subprograms are defined following the Var statement.

```
Function MOUSE (X: Interger) :Real;
    VAR
     I := integer;
Begin
     MOUSE := I - X;
End.                                   { MOUSE }
```

Functions take on values like variables, so they must have a declared type. *Procedures*, on the other hand, work like statements. They do not have a type, but rather they perform actions like any other statement in the instruction or body part of a program.

Procedures and functions communicate with the main program by way of a list of variables enclosed in parentheses. These are called *parameters*, and they make it possible for data to flow into and out of the subprogram.

A fully configured Pascal program contains constants, types, variables, and subprograms within its data declaration part. The optional subsections must occur in the following order:

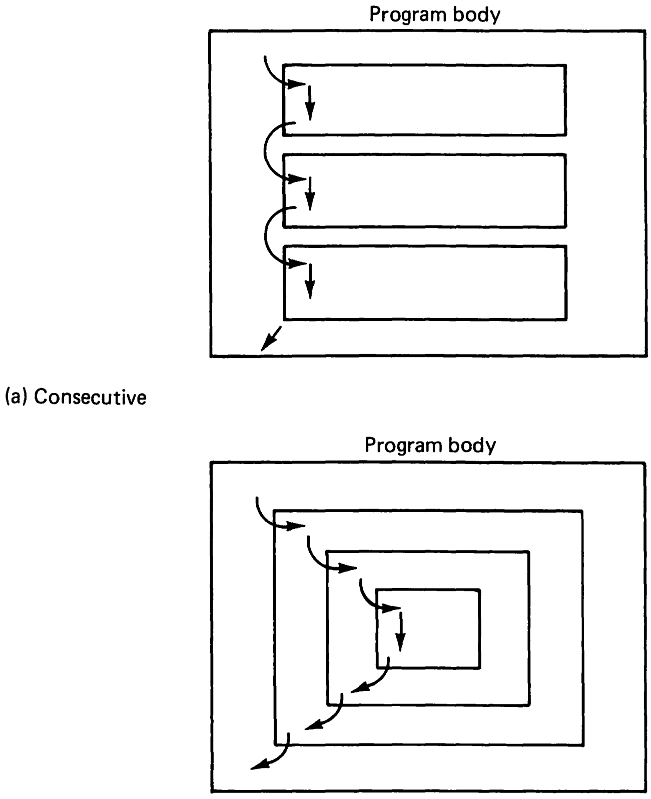
```
Program MAIN;
    Const                          { Const first }
           X = 10;
    Type                           {. . . followed by Type }
           Y = real;
    Var                            {. . . then Var . . . }
           Z :Y
    Function W : char;             {. . .and subprograms }
           begin
             :
           end;
Begin
               { program body }
End.
```

Body Part

The *body part* contains the instructions to be carried out when the program "runs," and it is enclosed between a pair of reserved words: Begin and End.

```
Begin                          { start instructions }
      { executable instructions }
End.                           { end of body, and a period }
```

The instructions of Pascal belong to one of three classes: sequence, looping, and decisions. These three allow you to write any program by combining statements in two fundamental ways: (1) consecutive statements are done one following the other, and (2) nested statements are done from the outer level to the inner level, and then out again.



(a) Consecutive

(b) Nested

FIGURE A.1
Consecutive versus nested instructions.

The difference between consecutive and nested statement structure in Pascal is shown graphically in Figure A.1. A nested structure resembles boxes stacked on top of one another. A Pascal program contains both nested and consecutive statements.

Sequence

Assignment, procedure calls, and compound Begin-End statements belong to this group. Here is an example of an assignment statement and procedure-call sequence.

```
Begin                              { program body }
      A := B + C;
      WriteLn (A, B, C);
End.
```

The Begin-End pair is used to nest one level of statements within another layer. The following example is exaggerated, but legal nonetheless.

```
Begin                     { program body }
   Begin                     { level 1 }
      Begin                     { level 2 }
         Begin                     { level 3 }
                  A := B + C;
         End;                      { level 3 }
      End;                      { level 2 }
   End;                      { level 1 }
End.                      { program body }
```

Typically, nesting occurs in combination with other (sequence) statements. The following examples show more realistic uses of Begin-End.

Looping

A *loop* is any section of a program which is repeatedly executed. Pascal has three looping statements. *Repeat-Until* checks for loop termination at the end of the loop; *While-Do* tests for termination at the beginning of the loop, and *For-Do* repeats the loop a specified number of times.

```
Repeat
          SUM := Sum + X;
          N    := N + 1
Until N > 10;
```

This Repeat loop continues to execute the two assignment statements until "N > 10" becomes True.

```
While N < 10 Do
      begin
            SUM := SUM + X;
            N    := N + 1
      end;
```

This While loop first tests "N < 10" and then performs the loop as long as "N < 10" is True. The test is repeated before each iteration (loop). Notice the use of a Begin-End nested level in the While-Do loop versus the absence of the Begin-End pair in the Repeat-Until statement. This is an inconsistency in Pascal, but one which you must remember. The While-Do statement iterates a single (compound) statement, whereas the Repeat-Until statement can iterate one or more statements.

```
For I := 1 To N Do
      begin
            SUM := SUM + X
      end;
```

The *For* loop simply repeats for as long as "I" equals 1,2,3, . . . N. The value of "I" is automatically incremented (by one) each pass through the body of the loop. Again, notice the use of Begin-End nesting if two or more statements are to be iterated. (This example could have been written without the Begin-End reserved words because only one statement was to be iterated.)

Decisions

There are two kinds of decision statements in Pascal. The *If–Then–Else* statement is used to select one path to be followed by the program from among two choices. The *case* statement is used to select one path to be followed from among many choices.

```
If A < B
      Then A := 0
      Else B := 0;
```

This simple If-Then-Else statement tests the truth of "A < B", and if it is True, the *Then* clause is executed. If the test is False, the *Else* clause is executed. Either "A := 0" or "B := 0" is performed; not both.

The If–Then–Else statement works for all types of data, but it is capable of making only a two-way choice. The case statement can make a multi-way choice but is restricted to making decisions on character, integer, and Boolean (logical variables valued only as true or false) data types only.

```
CASE X Of
        'A'  :  WriteLn ('a');
        'B'  :  WriteLn ('b');
        'C'  :  WriteLn ('c');
End;
```

Here the type of X must be character, and which WriteLn statement is executed depends on the value of X. If X is 'A', 'B', or 'C', the appropriate case clause is executed, but if X is none of the three choices, the case statement has no defined action (a BUG!).

An Example

The following example summarizes the features of Pascal discussed in this brief overview. Pascal contains many more features than surveyed here; this serves to introduce the overall nature of Pascal rather than as a reference. For the complete story, work through the sessions.

```
Program MAIN;
   Const
            N = 10;
   Type
            DATA = Integer;
   Var
            COUNT : 1 . . . N;
   Function SUM(M:Integer) : DATA;
          Var
                I    : Integer;
                S,Y  : DATA;
          begin
                S := 0;
                For I := 1 To M Do
                  begin
                       ReadLn (Y);
                       S := S + Y
                  end;                      { For loop }
                SUM := S;
          end;                              { Function Sum }
   Begin       { program body }
               Repeat
                       Write('Enter COUNT = ');
                       ReadLn(COUNT)
               Until   (COUNT <= N);
               WriteLn ( SUM(COUNT) )
   End.        { program MAIN }
```

This example contains a single function subprogram with its own data and instructions. The body of SUM computes the sum of M numbers entered from the keyboard.

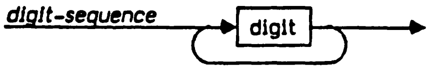
The body of the main program iterates until a number less than or equal to N is entered from the keyboard. Then the value of SUM is computed and displayed on the screen.

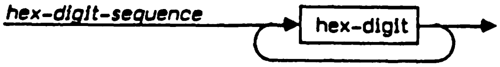
N is 10, and COUNT is an integer between 1 and N, inclusively. M is an input parameter to SUM, and SUM is the output returned from function SUM. S is simply a working variable used to accumulate the running total of numbers entered through variable Y. Similarly, I is a working counter within SUM.

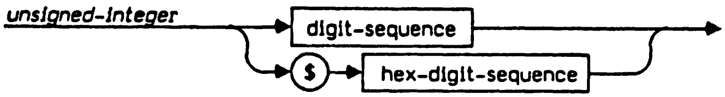
Appendix B

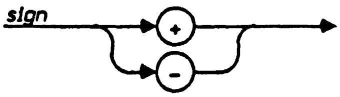
Pascal Syntax Diagrams

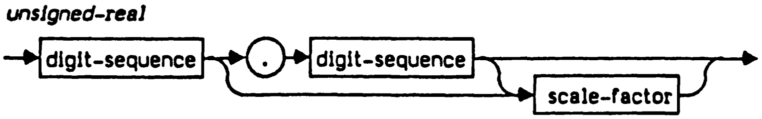
Numbers

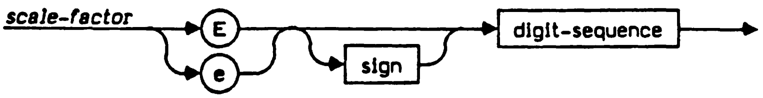


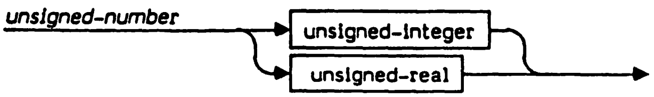


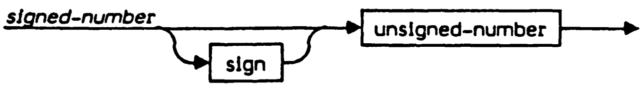




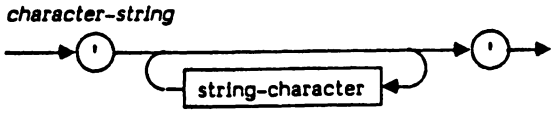


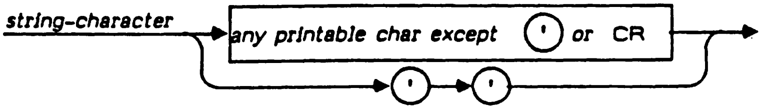




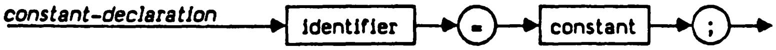


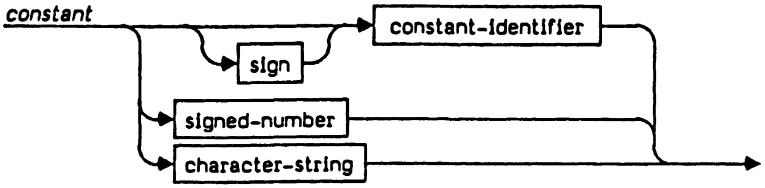
Character-Strings

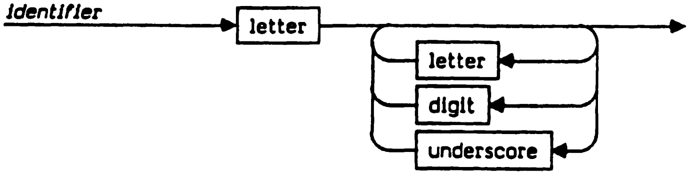




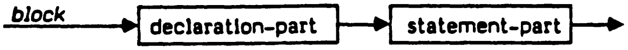
Constant-Declarations

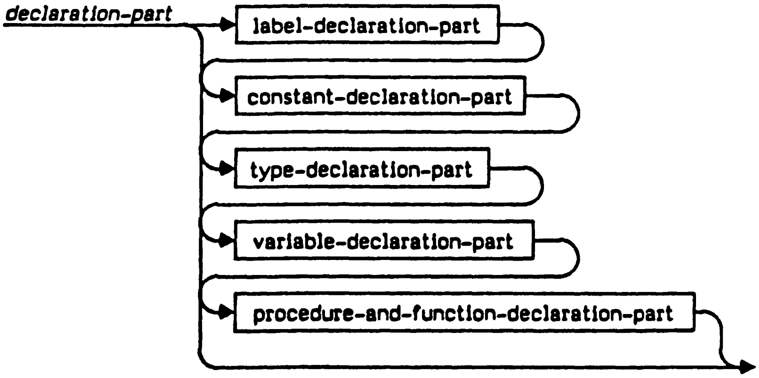


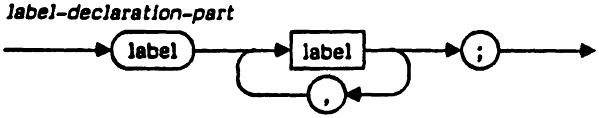


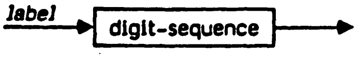


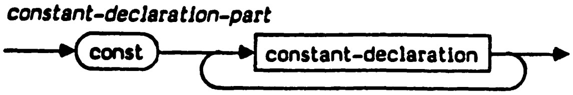
Definition of a Block

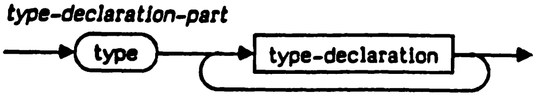


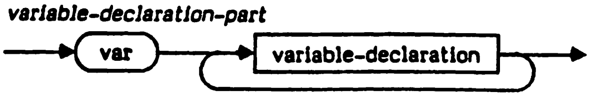

declaration-part
label-declaration-part
constant-declaration-part
type-declaration-part
variable-declaration-part
procedure-and-function-declaration-part

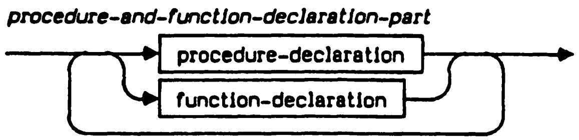


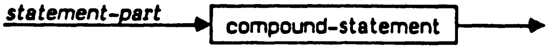




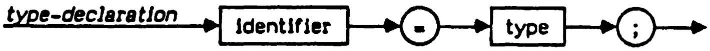


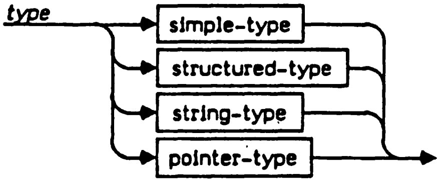




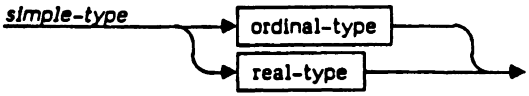


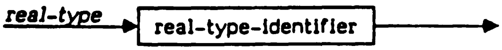
Types

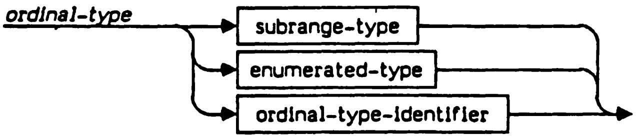




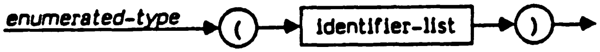
Simple-Types

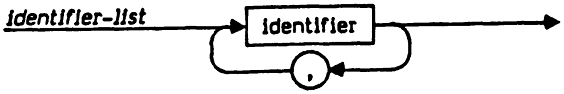




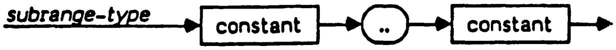


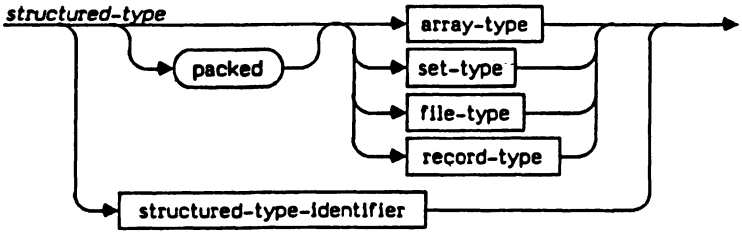
Enumerated-Types



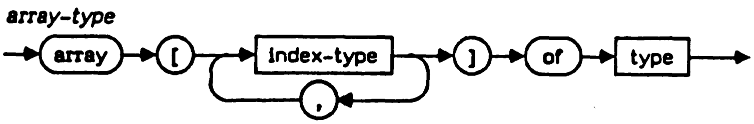


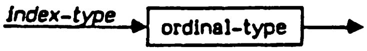
Subrange-Types



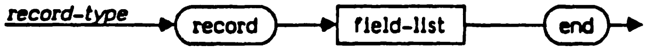


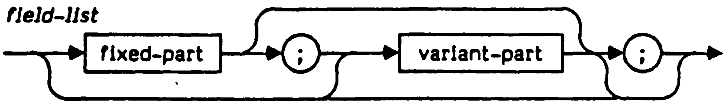
Array-Types

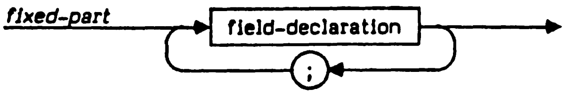


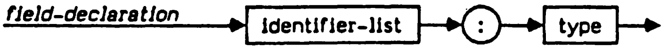


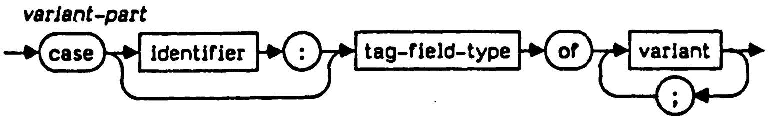
Record-Types

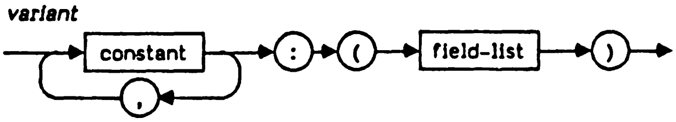


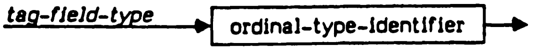




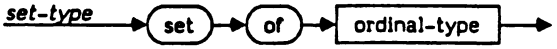




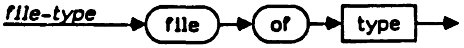




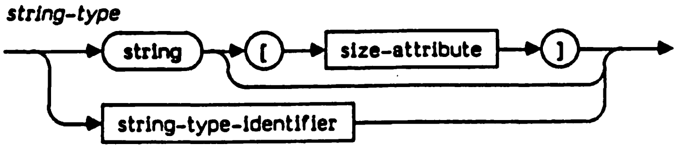
Set-Types

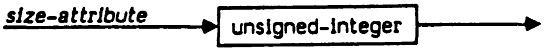


File-Types

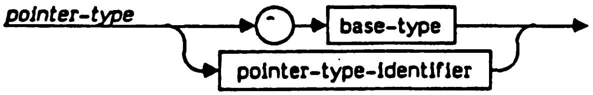


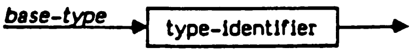
String-Types



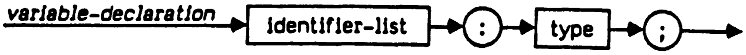


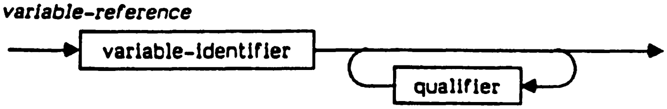
Pointer-Types

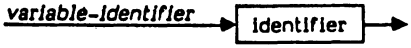


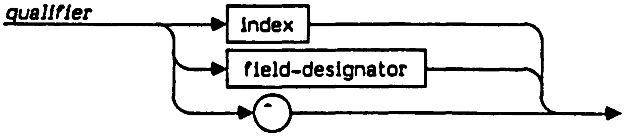


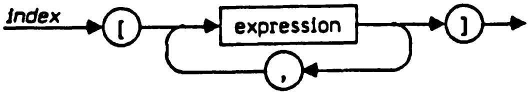
Variables

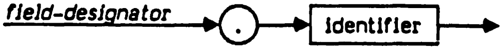

variable-declaration
identifier-list
:
type
;


variable-reference
variable-identifier
qualifier

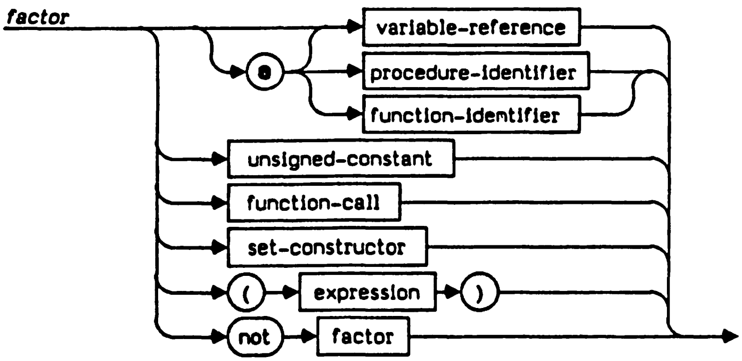

variable-identifier
identifier

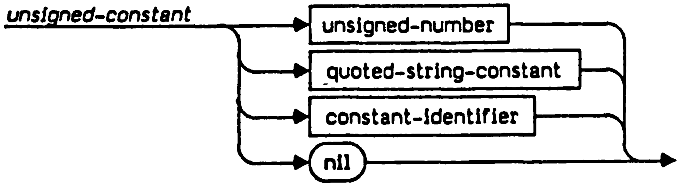

qualifier
index
field-designator
^

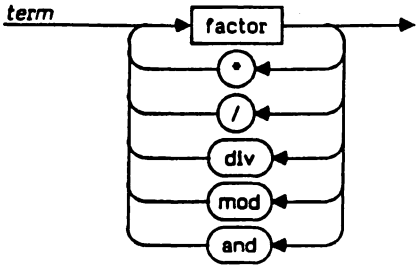

index
[
expression
,
]

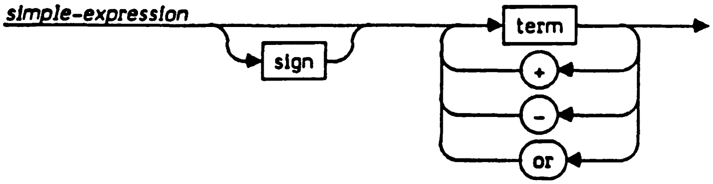

field-designator
.
identifier

Expressions

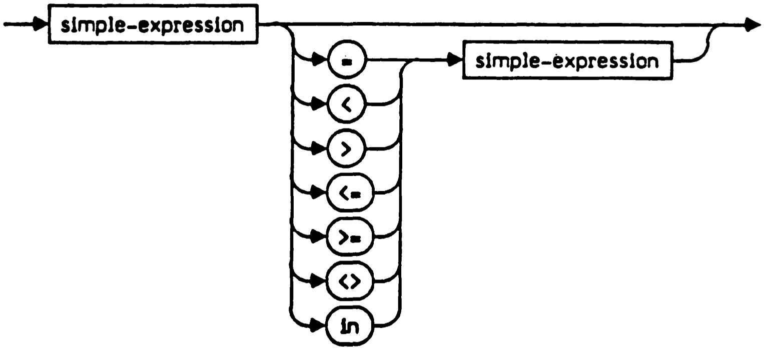




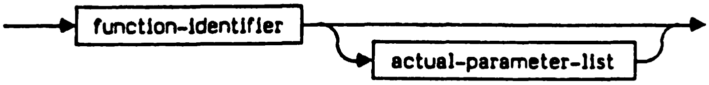




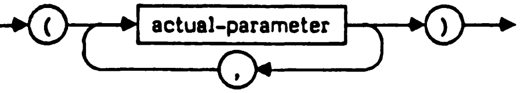
expression



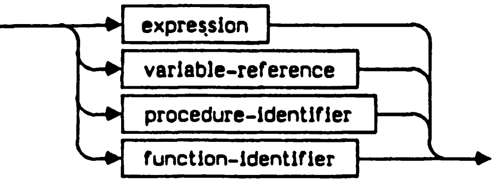
function-call



actual-parameter-list

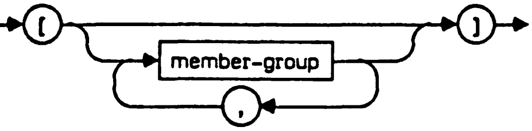


actual-parameter

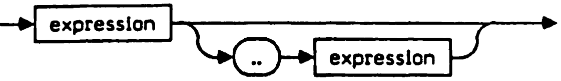


Set-Constructors

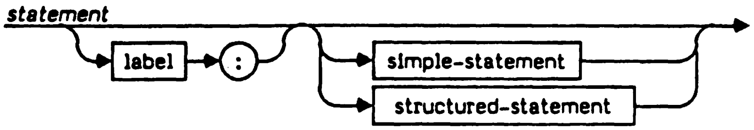
set-constructor

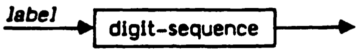


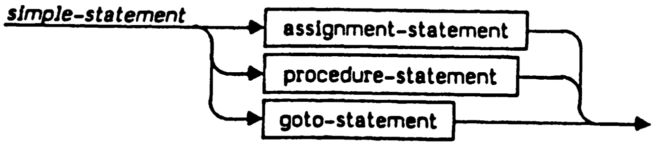
member-group

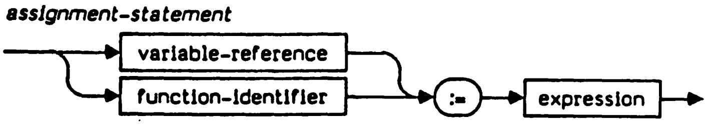


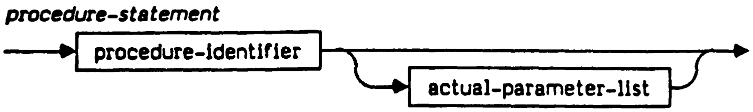
Statements

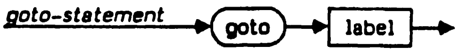

statement
label
:
simple-statement
structured-statement

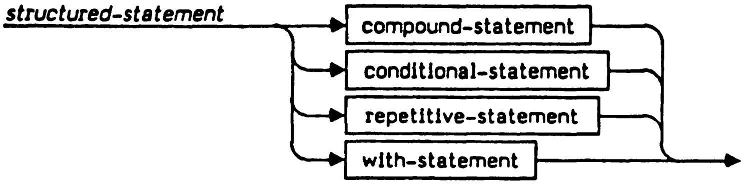

label
digit-sequence

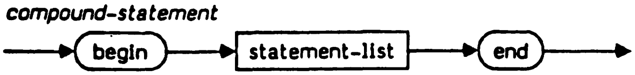

simple-statement
assignment-statement
procedure-statement
goto-statement

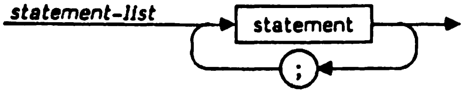

assignment-statement
variable-reference
function-identifier
:=
expression

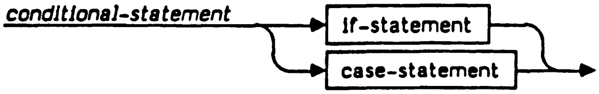

procedure-statement
procedure-identifier
actual-parameter-list

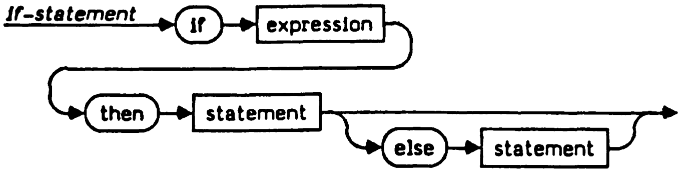

goto-statement
goto
label

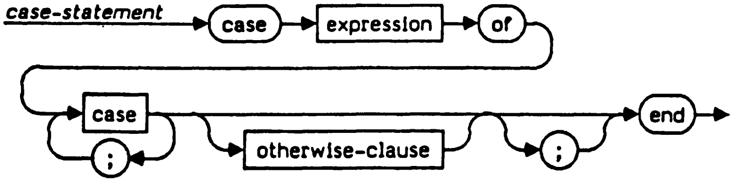

structured-statement
compound-statement
conditional-statement
repetitive-statement
with-statement

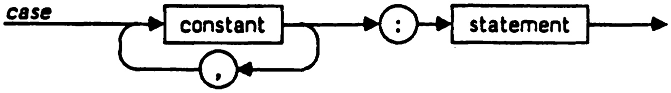

compound-statement
begin
statement-list
end

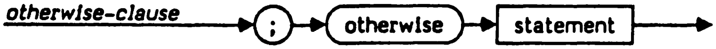

statement-list
statement
;

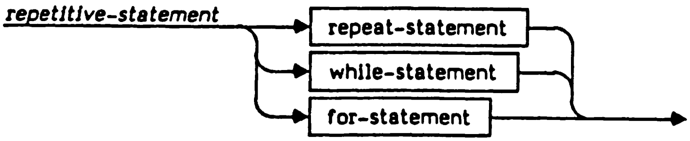

conditional-statement
if-statement
case-statement

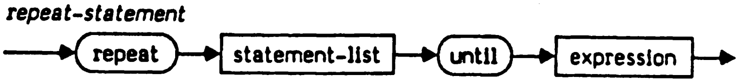

if-statement
if
expression
then
statement
else
statement

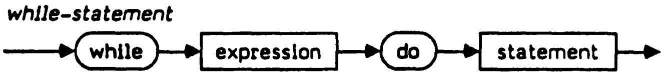

case-statement
case
expression
of
case
;
otherwise-clause
;
end

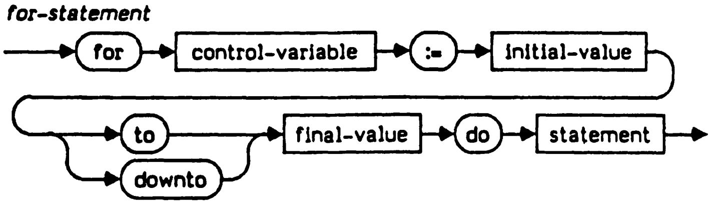

case
constant
,
:
statement

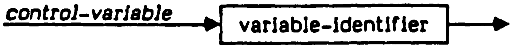

otherwise-clause
;
otherwise
statement

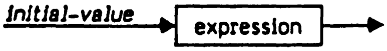

repetitive-statement
repeat-statement
while-statement
for-statement

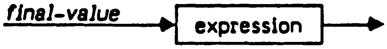

repeat-statement
repeat
statement-list
until
expression

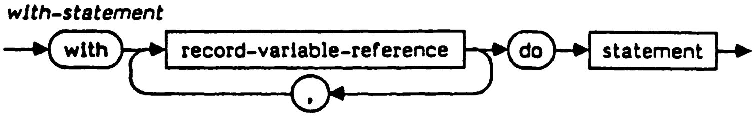

while-statement
while
expression
do
statement


for-statement
for
control-variable
:=
initial-value
to
downto
final-value
do
statement

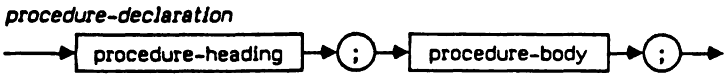

control-variable
variable-identifier

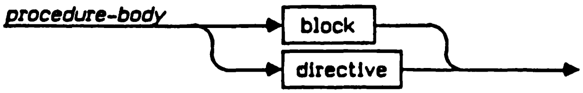

initial-value
expression

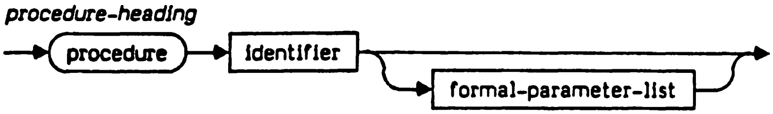

final-value
expression

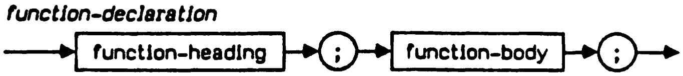

with-statement
with
record-variable-reference
,
do
statement

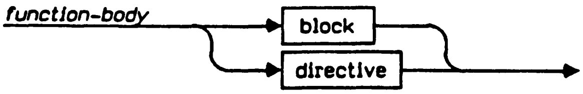
Procedures and Functions

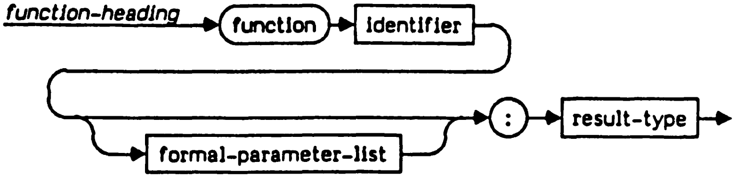


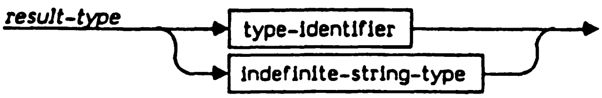


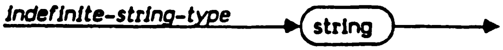


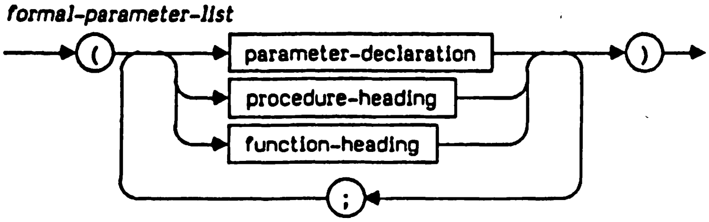


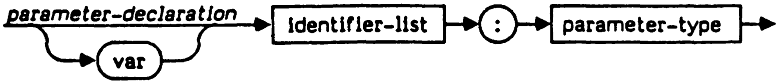


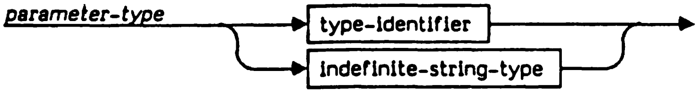




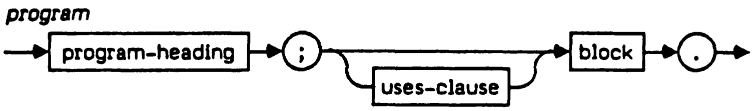


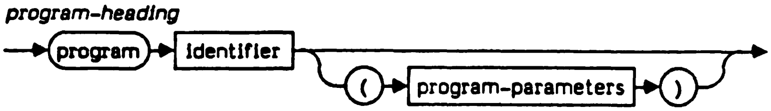

formal-parameter-list
(
parameter-declaration
procedure-heading
function-heading
)
;

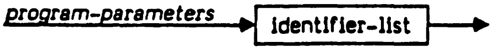

parameter-declaration
var
identifier-list
:
parameter-type

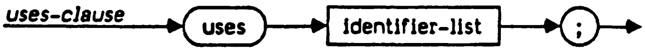

parameter-type
type-identifier
indefinite-string-type

Programs


program
program-heading
;
uses-clause
block
.


program-heading
program
identifier
(
program-parameters
)


program-parameters
identifier-list


uses-clause
uses
identifier-list
;

Appendix C

Debugging Macintosh Pascal Programs

Throughout this book you have used STOPS-IN, Observe, or Instant Windows to either slow down or stop the execution of your programs so you could examine the effect of the running program on one or more statements. These techniques are debugging tools, *and they come in handy for finding bugs in programs.*

In general, the nature of bugs can be so different from one program to another that categorizing them is not a simple task; a solution to one bug may not work for another bug. Nevertheless, the following list of common bugs may be used as a general guide to finding program errors and fixing them.

Where to Look for Bugs

1. Certain variables are not set to the correct values.
2. There are one or more infinite loops.
3. A flow control problem exists, and the segments in which the operations are supposed to be performed are not correct.
4. A bug exists in the Pascal Interpreter itself.

You should consider the last possibility very unlikely and not jump to the conclusion that since your program is not working, there is a bug in the Interpreter. Pascal has been tested thoroughly before being released for public use.

Possibility 1: Variables Not Set to Correct Values.

The following is a partial list of cases that may cause this error.

1. You have forgotten to initialize one or more variables.
2. If their values were read in:
   a. They were not read correctly.
   b. The contents of the file from which they were read are not correct.
   c. You may not have read them at all.
3. If they are to be set by a procedure call:
   a. Their corresponding procedure parameters may be defined as pass-by-value instead of pass-by-reference.
   b. They are not changed in the procedure.
4. A side-effect from one or more procedures and functions altered the value of one or more variables.
5. A scope rule is not understood and the variable is not defined in the block you are using it in.

Possibility 2: The Program Contains an Infinite Loop.

A partial list of cases that may cause this error is as follows:

1. Loop variables are not set to correct values.
2. The terminating condition is not set within the loop:

```
STOP := False;
While (not STOP) Do
 begin
   *
   *
   *
 end;
```

3. Loop index variables are modified within the loop, so that they can never reach a termination value.
4. Due to a flow control problem, the loop termination condition cannot be reached.

Possibility 3: Flow Control Problem

The following is a partial list of cases that may cause this error:

1. Boolean variables and expressions are not computed correctly.

2. The formation of a Boolean expression may not be correct:

```
If A < B Then . . . .
```

when what you meant was:

```
If A > B Then . . . .
```

3. Incorrect use of AND, OR, or NOT:

```
Not (A and B)
```

is incorrectly written as:

```
(Not A) and (Not B)
```

Instead of:

```
(Not A) or (Not B)
```

Possibility 4: Pascal Interpreter Has a Bug

The Interpreter as a program may have problems which cause failure in a correct program. But as stated earlier, this possibility should be ruled out as long as all the other possibilities have not been ruled out.

How to Locate a Bug

The general approach to locating a bug is by "divide and conquer." Find the portion of the program causing the bug by checking and eliminating one piece of the program at a time.

Check the parameter declarations of all procedures, the scope rules of all variables, and make sure there are no side effects of any kind.

Next use STOPS-IN to trace the flow of control of the program to make sure statements are executed in the correct order. Use the Observe Window to verify the values of certain variables, both global and local. You may want to stop the program at certain critical places and examine the value of important expressions using the Instant Window. In particular, examine control variables, variables used in conditional expressions, and the value of global variables or variables passed as parameters. Also, make sure the terminating conditions for While and Repeat-Until loops are stated correctly and that they are initialized correctly.

Appendix D:

Error Messages

1. This doesn't make sense

   A very general error message which will be triggered only when no other specific error can be identified.

2. The name "<name>" has already been defined at this level.

   Any attempt to re-declare an identifier within a block will elicit this message.

3. An invalid variable, field, or formal parameter list definition is found. A colon might be missing.

   A general error message which will be triggered by any attempt to use an undefined identifier. {"Undefined name" in Observe}

4. The name "<name>" has not been defined yet.

   A general error message which will be triggered by any attempt to use an undefined identifier. {"Undefined name" in Observe}

5. A type or procedure name has been found where a variable, field name, or value is required.

   A general error message which covers this situation in many contexts. {"Wrong kind of name" in Observe}

6. A type is expected. "<name>" is defined, but not as a type.

   A general error message which covers this situation in many contexts.

7. A constant is expected. "<name>" is defined, but not as a constant.

   A general error message which covers this situation in many contexts.

8. A subrange boundary has been found whose type is not integer, char, or enumerated.

   A specific error message that can arise in any subrange definition. Note that LongInt is excluded as a subrange host type.

9. A subrange has been found whose boundaries are not of the same type.

   A specific error message that can arise in any subrange definition.

10. A subrange has been found whose lower boundary is greater than its upper boundary.

    A specific error message that can arise in any subrange definition.

11. An array index has been found whose type is not integer, char, enumerated, or subrange.

    A specific error message that can arise in any array declaration. Note that LongInt is excluded as an array index type.

12. An invalid enumerated list has been found.

    A very general error message that can occur whenever an enumerated type is defined explicitly or implicitly.

13. A semicolon (;) is required on this line or above but one has not been found.

    A very general error message that can occur wherever a semicolon and only a semicolon is required (i.e., an End won't work).

14. Did not find a valid result type in the heading of this function's definition.

    Occurs whenever the result type of a function declaration is syntactically incorrect.

15. A colon (:) is required on this line or above but one has not been found.

    A very general message that can occur whenever a colon has been omitted. A colon is required after a statement label, after a case constant, and before the type of a function, variable, parameter, or a field declaration.

16. Either a semicolon (;) or an Until is expected following the previous statement, but neither has been found.

    A specific error message that can occur only in the context of an open repeat statement.

17. Either a semicolon (;) or an End is expected following the previous statement, but neither has been found.

A specific error message that can occur only in the context of an open compound statement or case statement.

18. An invalid Program parameter list has been found.

    A general error message that can arise whenever any syntactical error is discovered in the parameter list of a program heading.

19. Uses can only appear immediately following the Program heading.

20. A variable or function name is expected. "<name>" is defined, but not as a variable or function.

    A general error message that can arise whenever the name to the left of an assignment is defined to be other than a function name or variable name.

21. A period (.) is required following the last End of the program but one has not been found.

22. A type is required to complete a definition on this line or above but one has not been found.

    A very general error message that can occur whenever a type is required but not found.

23. An invalid formal parameter list has been found.

    A general error message that can arise whenever any syntactical error is discovered in the formal parameter list of a procedure or function declaration.

24. An End is required to complete the record definition above but one has not been found.

25. A field name is expected. "<name>" is defined, but not as a record.

    {"Undefined field name" in Observe}

26. A record name is expected. "<name>" is defined, but not as a record.

    {"Undefined record name" in Observe}

27. A Case constant is required here but one has not been found.

    A general error that will arise whenever a case statement lacks at least one case constant.

28. An invalid variant definition has been found.

    A general error that can arise whenever any syntactical error occurs in a variant declaration.

29. The size of this String should be a number between 1 and 255 but it is not.

30. A set should have elements whose type is integer, char, or enumerated but this set does not.

This error can arise whenever an attempt is made to construct a set whose base type is other than integer, char or enumerated. Note that LongInt is excluded as the base type of a set.

31. The name "<name>" doesn't make sense here.

    A general error that arises when an identifier that is not a procedure name appears in statement context.

32. This label has not been defined.

33. A Program keyword was not found at the beginning of this program.

34. This statement or keyword doesn't belong here.

    A general error message that will arise when statements appear in a declaration context or declaration parts are out of sequence.

35. This kind of declaration doesn't belong here.

    A very general error message that arises whenever a declaration appears in an inappropriate declaration part.

36. At least one constant declaration is required after the keyword Const, but none has been found.

37. At least one variable declaration is required after the keyword Var, but none has been found.

38. At least one type declaration is required after the keyword Type, but none has been found.

39. "End." is required at the end of a program, but it was not found.

40. At least one library name is required after the keyword Uses, but none has been found.

41. An invalid library name has been found.

42. This is not allowed in the Instant Window.

    A general error message that will arise whenever there is an attempt to use declarations in the Instant Window.

43. The value of this constant is not numeric and may not have a sign.

    A very specific error message covering the situation where an attempt is made to put a sign before a nonnumeric value in a constant declaration.

44. This does not make sense as a statement.

    A very general error message that can arise whenever something that is not a valid statement is encountered in a statement context.

45. The available memory for variables defined at this level has been exhausted.

    A fairly specific error message that will arise when a stack frame exceeds the implementation-defined limit.

46. This declaration or statement does not belong here.

    A fairly general error message that will arise whenever a statement or declaration or invalid directive immediately follows the heading of a procedure or function declaration.

47. A variable of this type would be too large.

    A fairly specific error message that will arise when the size of a declared type exceeds the implementation-defined limit.

48. Too many up-arrows are being applied to "<name>".

    {"Too many up-arrows" in Observe}

49. Too many indices are being applied to "<name>".

    A very general error message that can arise when more indices are used in an array variable reference than are defined for the variable. This error will also be issued when an attempt to index a non-array variable is made. {"Too many indices" in Observe}

50. This attempt to assign a result to the function named "<name>" outside of its definition is invalid.

    An assignment to a function name can only be done inside the function itself, in order to specify the value that the function returns.

51. This formal parameter type should be a named type or String, but is not.

52. This is an invalid variant selector.

53. This Case selector is not a valid expression.

54. An invalid list of variable names has been found.

    A very general error message that can occur whenever any syntactical error occurs in the declaration of a list of variables.

55. An invalid label was found on this line. A label must be a number between 0 and 9999.

56. A statement has already been labeled with this label.

57. The control variable in this For statement is invalid because it is defined outside of this procedure or function.

Errors While Editing

1. Excessive nesting of expressions, statements, or record has been detected at least once.
2. This statement is too long.
3. One or more comments containing more than 255 characters has been truncated.
4. One or more names or constants containing more than 255 characters has been truncated.

Appendix E:

Selected Solutions to Session Problems

Session 1

4. Drag a window to a new location on the desk top.
   Drag an icon around the desk top.
   Change the size of a window.
   Pulling down a menu.
   Selection of menu item by pulling down a window to the item to be selected and releasing the mouse button.
   Activating different windows or icons by clicking while pointing to them using the mouse.

Session 2

3. Locating the mouse pointer on the visible portion of the Text Window and clicking.
4. Nothing happens (the request is ignored).
5. Move the I-beam (insertion point) to the end of the first line: click the mouse; enter the line; then press the return key.

Session 3

1.

```
Var
  PI : Real;
```

7.

```
WriteLn ('IT'S TIME TO PARTY!');
```

8. The content of a variable can be altered during a program, whereas the content of a constant cannot.

9.

```
Program Add;
   Var
              B, C : Integer;
Begin
              ReadLn(B);
              ReadLn(C);
              WriteLn(B + C);
End.
```

Session 4

1. a. 20
   c. 4
   e. 1.95
   f. 5.0
   g. 4.0

3. a. 123 printed
   b. a BUG message appeared

4.

```
Program Cube:
            Var
                  S : Real;

       Begin
            Write('Enter a value for S: ');
            ReadLn(S);
            WriteLn('The AREA of a cube is: ', S * S * S:10:2);
       End.
```

5. a. 2.00 or 2.0 e+0 (depending on the Write statement)
   b. 36
   c. a

Session 5

```
1.  Procedure CALC_TAX (Var TAX, TAX_RATE, AMOUNT : Real);
    Begin
                TAX := TAX_RATE * AN.MOUNT;
    End;

4.  Procedure AREA_PERI (Var AREA : Real;
                             Var PERIMETER : Real;
                                 Length    : Real;
                                 Width     : Real);
    Begin
          AREA := Length * Width;
          PERIMETER := 2 * (Length * Width);
    End;
```

Because we need to calculate values, functions are good when only one value is to be returned.

Session 6

```
1.  If (Last >= ’A’) And (Last <= ’Z’)
          Or ( (Last > = ’a’) And (Last <= ’z’) ) Then
          Case Last of
          ’A’, ’a’, ’B’, ’b’, ’C’, ’c’, ’D’, ’d’, ’E’, ’e’ :
                WriteLn(’Tigers’);
          ’F’, ’f’, ’G’, ’g’, ’H’, ’h’, ’I’, ’i’, ’J’, ’j’ :
                WriteLn(’Lions’);
          ’K’, ’k’, ’L’, ’l’,’M’, ’m’, ’N’, ’n’,’O’, ’o’ :
                WriteLn(’Panthers’);
          ’P’, ’p’,’Q’, ’q’, ’R’, ’r’,’S’, ’s’, ’T’, ’t’ :
                WriteLn(’Leopards’);
          ’U’, ’u’, ’V’, ’v’, ’W’, ’w’, ’X’, ’x’,’Y’, ’y’, ’Z’, ’z’ :
                WriteLn(Sabres’);
     end
    Else
     "
     "
     "
```

5.
```
Program No5;
 Var
     Number : Integer;
 Begin
     Write ('Enter Number: ');
     ReadLn (NUMBER);
     Case NUMBER of
       1,2           :
          WriteLn ('No way!');
       5,3,13        :
          begin
            NUMBER := NUMBER + 1;
            WriteLn (NUMBER)
          end;
       7,11          :
          begin
            NUMBER := NUMBER - 1;
            WriteLn (NUMBER)
          end
     Otherwise
                WriteLn ('Wrong Input!');
     end; {Case}
 End. {Program No5}
```

Session 7

1. a. Computes:

P := 1 * 2 * 3 * 4 * 5
P = 120

c. Computes square of 1 . . 10

I * I

1 * 1 = 1^2 = 1

2 * 2 = 2^2 = 4

3 * 3 = 3^2 = 9

3

*

*

*

10 * 10 = 10^2 = 100

```
5.   Program GUESS;
       Const
          ANSWER = 67;
       Var
           i    : Integer;
          nog : Integer;
     Begin
          i := 0; nog := 0;
          While (i <> ANSWER) Do
               begin
                    Write('Enter your guess  ');
                    ReadLn (i);
                    If i > ANSWER Then
                      WriteLn('Next time enter a lower number.')
                    Else
                      If i < ANSWER Then
                         WriteLn('Next time enter a higher number');
                    nog := nog + 1;
               end;
          nog := nog + 1;
          WriteLn ('You made ',nog, ' guesses.')
     End.     {GUESS}
```

Session 8

```
1.   Procedure SUM (Var Feet2  : Integer;
                        Feet1  : Integer;
                   (Var In2    : INCHES;
                        In1    : INCHES) ;
           Var TotIn : Integer;
     begin
           TotIn := In1 + In2;
           Feet2 := Feet1 + Totin Div 12;
           In2   := Totin Mod 12
     end;     {procedure}

3.   Procedure TIC_TOC (Var HR:Hours; Var MIN : MINS; Var SEC : SECS);
     Var I,J : Integer;
     begin
           I     := SEC + 1;
           SEC := I MOD 60;
                   J     := MIN + I Div 60;
                   MIN := J MOD 60;
                   HR   := HR + J Div 60
     end;
```

Session 9

1. To Procedure COPY_KBD_TO_FILE, insert

```
                     WriteLn      (Line);
after : ReadLn      (Line);
```

5. 1

7. Procedure COPY_FILE_TO_FILE (Fname1 : String; Fname2 : String);

```
 Var
      InFile    : text;
      OutFile  : text;
      Line       : String;
 begin
      Reset (InFile, Fname1);                 {open input file}
      Rewrite (OutFile, Fname2);              {open output file}
      ReadLn (InFile, Line);
      While Not EOF (InFile);
            WriteLn (OutFile, Line);
            ReadLn (InFile, Line)
      end      {while}
 end:     {COPY_FILE_TO_FILE}
```

Session 10

3. Procedure DISPLAY (PHBook: Book; Last: Integer);

```
   Var
      i = Integer;
begin
        For i := 1 TO Last Do
            WriteLn (PHBook [i]);
end;    {DISPLAY}
```

5. b.

```
      For i := 1 To N Do
          List[i] := 0;
```

d.

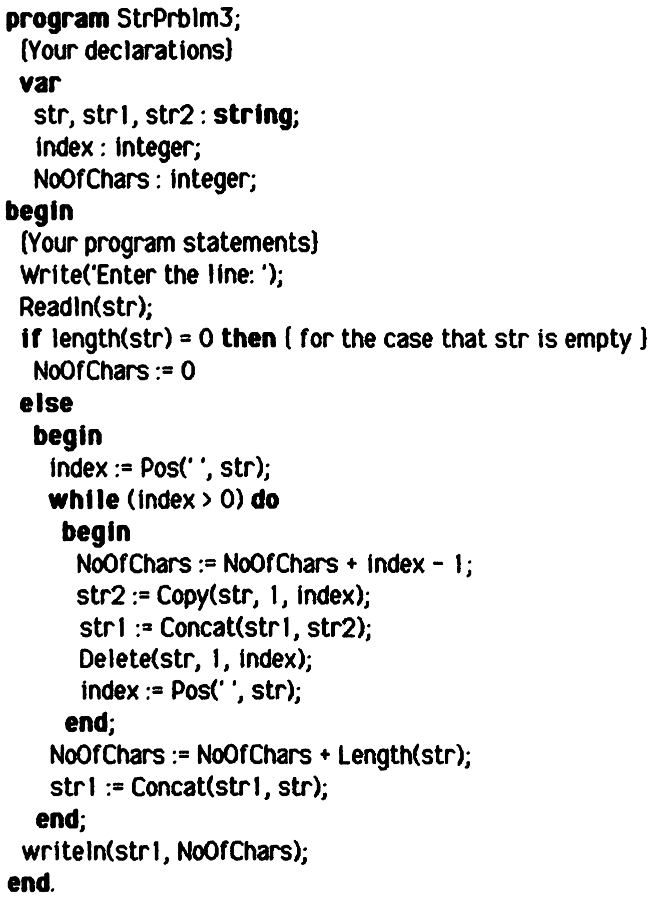
```
  j := 0;
  i := 2;
  While i <= N Do
     begin
          j         := j + 1;
          Out [ j ] := In[ i ];
          i         := i + 2
     end     {For}
```

g.

```
Count := 0;
For  i := 1 to ARRAY_MAX Do
          If LIST[i] > B Then
             COUNT := COUNT + 1;
 WriteLn  ('No. of array elements > B: ', COUNT);
```

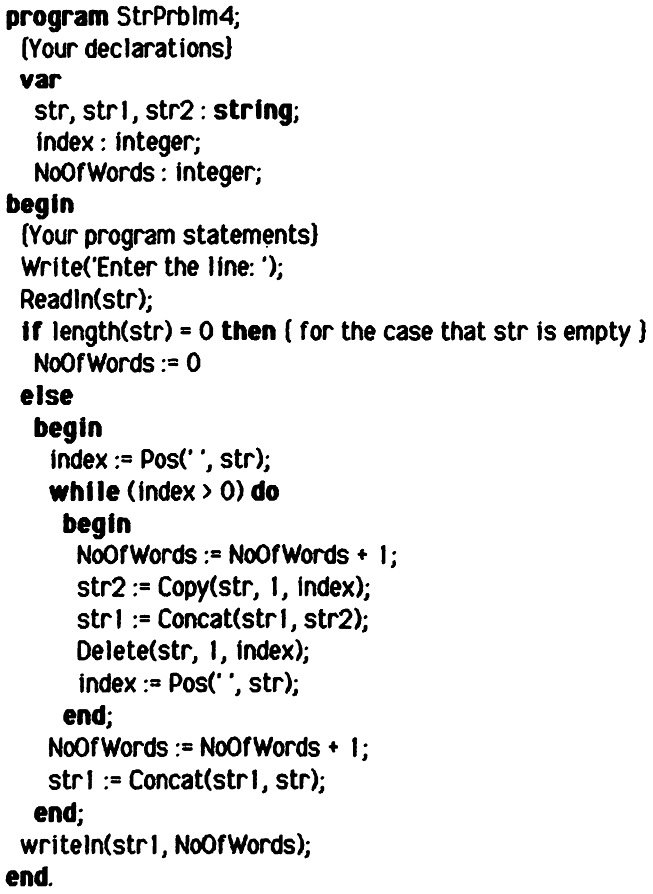
Session 11

3.



```
program StrPrblm3;
 {Your declarations}
 var
   str, str1, str2 : string;
   index : integer;
   NoOfChars : integer;
begin
 {Your program statements}
 Write('Enter the line: ');
 Readln(str);
 if length(str) = 0 then { for the case that str is empty }
   NoOfChars := 0
 else
   begin
     index := Pos(' ', str);
     while (index > 0) do
       begin
         NoOfChars := NoOfChars + index - 1;
         str2 := Copy(str, 1, index);
         str1 := Concat(str1, str2);
         Delete(str, 1, index);
         index := Pos(' ', str);
       end;
     NoOfChars := NoOfChars + Length(str);
     str1 := Concat(str1, str);
   end;
 writeln(str1, NoOfChars);
end.
```

4.



```
program StrPrblm4;
 {Your declarations}
 var
  str, str1, str2 : string;
  index : integer;
  NoOfWords : integer;
begin
 {Your program statements}
 Write('Enter the line: ');
 Readln(str);
 if length(str) = 0 then { for the case that str is empty }
  NoOfWords := 0
 else
  begin
   index := Pos(' ', str);
   while (index > 0) do
    begin
     NoOfWords := NoOfWords + 1;
     str2 := Copy(str, 1, index);
     str1 := Concat(str1, str2);
     Delete(str, 1, index);
     index := Pos(' ', str);
    end;
   NoOfWords := NoOfWords + 1;
   str1 := Concat(str1, str);
  end;
 writeln(str1, NoOfWords);
end.
```

Session 12

1. a.

```
Calendar =  Record
            Months : 1 .. 12;
            Weeks  : 1 .. 52;
            Days   : 1 .. 7
            end;     {record}
```

d.
```
PERSON = Record
           AGE    : 1 .. 99;
           SEX    : Char;
           Weight : 1 .. 400;
           Height : 1 .. 20
           end;     {record}
```

3.
```
S.A.   := 1;
S.T.B. := 'n';
S.T.C. := 2;
S.W.D. := 5;
```

4.
```
With A Do
   begin
        WriteLn (X);
        With Y Do
             WriteLn (B,C);
        WriteLn (Z)
   end;     {With}
```

Session 13

1. a. False
   b. True
   c. False

6. a.

   [], [1], [2], [3], [1,2], [2,3] [1,3], [1,2,3]

   d.

   [], [Red], [White], [Blue], [Red, White], [Red, Blue], [White, Blue], [Red, White, Blue]

7. Y can only take one value at a time, which is either A, B, or C. X can take more than one value up to 3 ([A,B,C]) or no value ([]). Altogether, X can have up to 8 different combinations of (A,B,C) as its value.

Session 14

1. a. Records in a sequential file must be accessed one at a time in sequence. To get record #5, the first four records must be read and discarded. In a random file, record #5 can be accessed directly—your program need only locate record #5 with a SEEK and then read the record.

c. RESET opens a file for input, thus a RESET file can be read only. REWRITE opens a file for output; hence a REWRITE file can be written only. Furthermore, if an existing file is opened with a REWRITE, its contents are lost!

4.
```
STUDENT = RECORD
            LastName   : string[20];
            FirstName  : string[10];
            Midterm1   : 0 .. 100;
            Midterm2   : 0 .. 100;
            Homeworks  : array[1 .. 5] of 0 .. 100;
            ClassStand : (Fr, Soph, Jr, Sr );
          end;
```

Session 15

3.
```
 1.4 e+4
 1.4 e+4
 1.7 e+4
-1.5 e+4
 1.9 e+4
 1.2 e+4
 1.1 e+4
```

6.
```
Function PI (N : Integer) : Integer;
begin
    If N = 1 Then
          PI := 1
    Else
          PI := N * PI (N - 1)
end;
```

Session 16

1. Add the following to the main program:

```
'O', 'o' : SAVE( PhoneList );
```

before the "otherwise" keyword. Then add the SAVE procedure at the beginning of your program.

```
procedure SAVE ( PhoneList: PagePointer );
  Type
   OutRec = record
             Name  : String20;
             Phone : String12;
        end;
```

```
Var
     FileName : string[8];
     OutFile    : file of OutRec;
begin
  WriteLn ( 'Enter output file name:');
  ReadLn ( FileName );
  REWRITE ( OutFile, FileName );
  while (PhoneList <> NIL ) do
       begin
        OutFile^.Name := PhoneList^.Name;
        OutFile^.Phone := PhoneList^.Phone;
        PUT ( OutFile);
        PhoneList :=  PhoneList^.NextPage;
       end;      {while}
CLOSE ( OutFile );
end;      {SAVE}
```

Note: It is assumed that the list contains at least one element already. Otherwise, an empty file will be created.

2. Add the following before the "otherwise" keyword in the main program.

```
'R', 'r' : READP ( PhoneList );
```

Then add the following procedure to your program.

```
procedure READP ( var PhoneList: PagePointer );
 Type
   InRec = record
     Name  : String20;
     Phone : String12;
    end;
 Var
   InFile     : file of InRec;
   FileName : string[8];
 Begin
   WriteLn ( 'Enter input file name:');
   ReadLn ( FileName );
   RESET ( InFile, FileName );
   while not EOF ( InFile ) do
     begin
       InsertPhone ( PhoneList, InFile^.Name, Infile^.Phone);
       GET ( InFile );
     end;      {while}
  end;      {READP}
```

Note: It is assumed that the file name is valid and that the file already exists.

Session 17

2.

```
program ConcertoM;
    (Modified version for problem 2 session 17)
 const
  MaxNote = 41; ( Max Note +1 )
 type
  Note = 1 .. 7;
  Tone = record
     cnt         : integer;
     Amplitude : integer;
     duration    : integer;
   end;
  SWYnthRec = record
     Mode : integer;
     Song  : array {1 .. MaxNote} of Tone;
   end;

var
 Freq : array [Note] of integer;
 Gperiod :  integer;
 StNote : array [Note] of string;
 Concert : SWYnthRec;
 Oct : real;
 i, NoteNo : integer;
 GDuration, GTempo, GOct, TicksP16Th, GAmplitude : integer;
 k : Note;
 Flat, Dot, Sharp : Boolean;
procedure  init;
begin
 Freq[1] := 264;
 Freq[2] := 297;
 Freq[3] := 330;
 Freq[4] := 352;
 Freq[5] := 396;
 Freq[6] := 440;
 Freq[7] := 495;
 StNote[1] := 'C';
 StNote[2] := 'D';
 StNote[3] := 'E';
 StNote[4] := 'F';
 StNote[5] := 'G';
 StNote[6] := 'A';
```

```
  StNote[7] := 'B';
  GOct := 4;
  GAmplitude := 100;
  Gtempo := 75;
  TicksP16TH := 900 div GTempo;
  Gduration := TicksP16TH * 16; {Whole Duration}
  Gperiod := 1;
end;
function  Octave (N : note;
                  OctNum : integer) : real;
var
  O, m : integer;
  Temp : real;
  begin
  if OctNum = 4 then
  Octave := Freq[N]
  else if OctNum < 4 then
  begin
    O := 4 - OctNum;
    Temp := Freq[N];
    for m := 1 to 0 do
      Temp := Temp / 2;
    Octave := Temp;
  end
else
  begin
    O := 4 - OctNum;
    Temp := Freq[N];
    for m := 1 to 0 do
      Temp := Temp * 2;
    Octave := Temp;
  end;
end;
function Convert ( str : string ) : integer;
  var
    i, len, No : integer;
    str : char;
```

```
begin
  len := length(str);
  No := 0;
  for 1 := 1 to len do
    begin
      str1 := copy(str, i, 1);
      No := ord(str1) - ord('0') + No * 10;
    end;
  Convert := No;
end; { Convert }
function FindNote (str : string) : Note;
  var
    index : Note;
    found : boolean;
    begin
    {Note: we are assuming the note exists for sure}
      found: := false;
      index := 1;
      while not found do
        begin
          if StNote[index] = str then
           begin
            FindNote := index;
            found := true;
           end
          else
           index := Succ(index);
        end;
    end; { FindFreg}
    function Count (Frequency : real) : integer;
    begin
      Count := round(783360 / Frequency);
    end;
    procedure FndDotShF1 (var str : string );
      var
        str1 : string ;
```

```
begin
  str1 := copy(str, 1, );
  Sharp := false;
  Dot := false;
  Flat := false;
  if str1 ='#' then
    begin
      Sharp := true;
      delete(str, 1 1);
    end
  else if str1 = '_' then
  begin
    Flat := true;
    delete(str, 1 1);
  end;
    str1 := copy(str, length(str), 1);
    if str1 = '.' then
  begin
    delete(str, length(str), 1);
    Dot := true;
  end;
  procedure BuildNote (str : string );
   var
    str1, str2 : string ;
    temp, period : integer;
    frq     : real;
    index : Note;
  begin
   str1 := copy(str, 1, 1);
   index := FindNote(str1);
   if length(str) = 1 then
    begin
     with Concert do
      begin
       Song[NoteNo].Duration := GDuration div Gperiod;
       Song[NoteNo].Amplitude := GAmplitude;
       Song[NoteNo].cnt := Count(Octave(index,      GOct));
      end
    end
  else
    begin
      delete(str, 1, 1);
      FndDotShF1(str);
      if length(str) > 0 then
      period := Convert(str)
    else
     period := Gperiod;
    frq := Octave(index, Goct);
    if Flat then
     frq := frq      * 24 / 25
    else if Sharp then
     frq := frq * 25 / 24;
    Temp := Gduration div period;
    If Dot then
     Temp := Round(Temp * 1.5);
    with Concert do
      begin
       song[NoteNo].Duration := Temp;
       Song[NoteNo].Amplitude := GAmplitude;
       Song[NoteNo].cnt := Count(frq);
      end
    end
  end; { of Build Note }
procedure RdNotes;
  var
    done : Boolean;
    str, str1 : string ;
begin
  writeln('To End Entering the Notes Type End for the Note');
  Writeln;
  wrintln('Enter the Notes      :      ');
  done := False;
  NoteNo := 1;
  repeat
    readln(str);
    if (str = 'end') or (str = 'END') then
     begin
      done := true;
      with Concert do
```

```
   begin
    Song[NoteNo].Duration := GDuration div Gperiod;
    Song[NoteNo].Amplitude := GAmplitude;
    Song[NoteNo].cnt := 0;
   end
  end
else
  begin
   str1 := Copy(str, 1, 1);
   if str1 = 'O' then
    begin
     str1 := Copy(str, 2, 1);
     GOct := Convert(str1);
    end
   else if Copy(str, 1, 2) = 'LN' then
    Gperiod: = Convert(copy(str, 3, length(str)))
   else
    begin
     BuildNote(str);
           NoteNo := NoteNo + 1
         end
       end
    until done;
   end;
   procedure Play;
   begin
    Concert.Mode := -1;
    StartSound(3Concert, SizeOf(Concert), Pointer(-1));
   end;
    begin
     {Your program statements}
     init;
     RdNotes;
     Play;
    end.
```

Session 18

The solutions to this session are too long to present here. You should consult the graphics reference card for help in doing these problems.

Index/Glossary

@, *241*
An operator that extracts the address of an object.

active window, *5*
A window that is active; its controls are activated and it is on the "top" of the desk.

actual parameters, *59*
The list of function or procedure parameters given when the function or procedure is called.

algorithm ii, *81*
A procedure for doing a calculation in an unambiguous, mechanical manner.

AND truth table, *65*

array, *111*
A collection of elements.

ASCII character set, *49*

ASCII code, *49*
The American Standard Code for Information Interchange.

assignment, *32*
The process of copying a value into a variable.

assignment operator, *33*

assignment operator *(continued)*
The assignment operator in Pascal is :=.

base type, *93*

Begin, *415*
A reserved word in Pascal to designate the beginning of a program or a section within a program.

binary number, *ii*
A string of ones and zeros, when interpreted by a computer, represent characters, numbers, pictures, and instructions.

block activation, *179*
The proces of running a block.

block-structure, *175*
The feature of Pascal that allows program parts to be connected together as either separate blocks, or nested blocks.

blocks, *175*
A section of a Pascal program containing its own data and instructions.

boolean, *31*
A type whose values can be either TRUE or FALSE, only.

boolean constant, *63*
TRUE or FALSE.

boolean variable, *63*
A variable whose value can only be TRUE or FALSE.

boolean, operations on, *64-66*

bug dialog, *18*
A dialog box which appears whenever your program cannot continue due to a programming error.

bugs, *271-273*
An error in a program.

buttons, *20*
A dialog box has special selection boxes called buttons which are selected by clicking them. For example, OK and CANCEL are buttons.

case clause, *74*

case selector, *74*
The ordinal value used to determine which case dause is executed.

case statement, *74*
definition *77*

char, *31*,
operations on *49-50*
A type whose values can be a single letter, numeral, or keyboard character.

CHR, *52*

click-hold, *3*
The act of pressing the mouse button and not releasing the button.

clicking, *2*
The act of pressing and releasing the button on the mouse.

close box, *5*
A small rectangular region in the upper left-hand corner of a window (in the control bar), that closes the window when it is clicked.

ClosePicture, *243*

closing a window, *6*
See close box.

comments, *40*
Statements that are ignored by the Pascal interpreter, but are used by programmers to document the program.

compiler, *12*
A language translator that converts a high-level language program into machine language prior to running the program. Contrast with interpreter.

compound if clause, *70*

CONCAT, *127*

consecutive statements, *250*

Const, *31*, *92*
The reserved word that designates the list of constants used by a program.

constant, *30-31*
Objects which cannot change in value are called constants *See* also Const.

control bar, *5*
A region at the top of a window that is used to close the window or move the window by clicking or dragging.

COPY, *127*

CRT, *1*
A cathode-ray-tube; the display screen of a computer.

cursor, *2*
An arrow, vertical line, or other graphical image which indicates the current position of a mouse or keyboard character.

data declaration part, *247*

data structure, *137*
A logical structure for organizing information so that it can be processed by a machine.

data type, *91*

debugging tools, *271*
Techniques for finding and

debugging tools *(continued)*
eliminating errors in programs.
decisions, *250*
declaration part, *30*
The section of a Pascal program that defines all objects used by the program.
DELETE, *23, 127*
designator, *111*
An array subscript or index.
desk top, *i, 2*
The simulated top of a desk showing windows, icons, and dialogs: the metaphor used to make using the Macintosh easy and intuitive.
dialog, *8*
A special window that is used to communicate between the user and the program running on the Macintosh.
dimmed
A button, or menu item appears dimmed (in half-tone gray) when it is disabled—thus it cannot be selected.
direct access file, *100*
A file whose records can be directly accessed without reading the other records in the file.
disabled button, *20*
A dialog button that is not available for selection—it will appear as a dimmed button.
DISPOSE, *196*
dot notation, *143*
dragging, *3*
The act of moving the mouse while click-holding. Dragging is used to move graphical objects (icons, windows) or to select an item from a menu.
dragging a window, *7*
See size box or control bar.
drawing window, *10, 13, 231*
A window in the Macintosh Pascal interpreter program where the graphical output from a running program is displayed.

editing, *22*
The process of changing a program.
eject, *11, 21*
A diskette is removed from the disk drives by first ejecting it and then manually removing it. The File menu contains an EJECT item which must be selected to eject the selected diskette.
element, *111*
A single value or record within an array or data structure.
empty string, *124*
A string containing no characters, i.e., its length is zero.
enabled button, *20*
A dialog button that can be selected.
End, *41*
A reserved word in Pascal to designate the termination of a program or a section within a program.
enumerated scalars, *152*
EOF, *102*
End of File mark.
EOLN, *101*
End of Line.
external blocks, *185*
Procedures or functions which are defined outside the program that uses them.
external file name, *101*
The name of the file used by the diskette directory.

file. *99*
A collection of records stored on diskette.
file buffer, *103, 165*
file operations, summary, *173*
file records, *100*
The smallest unit of file storage.
file update, *169*
flow control, *272*
For loop, *85*
formal parameters, *59*

internal file name *(continued)*

Pascal program.

interpreter, *iv*

A language translator program that directly interprets the statements of some other (high-level) language.

iteration, *81*

A program iterates when a section of the program is repeated.

key, *120*

An element or portion of an element in a data structure which is used to identify, locate, or store an element in a list.

keywords, *27*

Special words in Pascal that cannot be used for anything except to construct syntactically correct statements in Pascal. *See* also reserved words.

LENGTH, *127*

linked list, *196*

A data structure containing elements linked together by pointers.

list, *115*

A data structure for arranging and storing information.

literal string, *124*

A string that is self-defined. A constant string.

locating bugs, *273*

locking

A file or entire diskette may be locked using the Get Info item under menu File in the desk top. Locked files and diskettes can not be written onto.

LongInt, *240*

loop, *82, 251*

An iteration in a program. *See* iteration.

loop condition, *82*

A condition that determines the number of times a loop repeats.

loop counter, *82, 85*

A variable that keeps track of the number of times a loop is repeated.

looping, *250*

machine language, *ii*

The sequence of binary numbers which directly control the electronic circuits of a computer.

Macintosh Pascal types (Table), *97*

manifest constant, *155*

A named constant, i.e. an identifier in Pascal that is assigned a constant value.

menu, *1, 3*

A list of options which can be selected from the desk top. A running program is controlled by its user through selections made from a menu.

menubar, *3*

A region at the top of the Macintosh screen where all menu titles appear.

modem, *105*

A device for communicating with another computer over the telephone network.

modular programming, *54*

A technique whereby a program is sub-divided into manageable parts called modules.

mouse, *1*

A hand-controlled device used to point and select items on the screen of a computer.

multidimensional programming, *12*

A style of programming in which multiple windows are used to simultaneously display many views of a program while it is being developed, tested, or run.

multiple windows, *12*

A desk top with two or more windows simultaneously displayed.

multiway branching, *73*

See also case statement.

music, *218*

QUICKDRAW GRAPHICS

T. G. Lewis, Abbas Birjandi

From *Macintosh™ Hands-On Pascal* by T. G. Lewis and Abbas Birjandi. Wadsworth Publishing Company, Belmont, CA.

Graphport

(GrafPort Routines)

(2) procedure InitGraf (globalPtr : QDPtr);
(2) procedure OpenPort (port : GrafPtr);
(2) procedure InitPort (port : GrafPtr);
(2) procedure ClosePort (port : GrafPtr);
(2) procedure SetPort (port : GrafPtr);
(2) procedure GetPort (var port : GrafPtr);
(2) procedure GrafDevice(device : Integer);
(2) procedure SetPortBits(bm : BitMap);
(2) procedure PortSize(width, height : Integer);
(2) procedure MovePortTo(leftGlobal, topGlobal : Integer);
(2) procedure SetOrigin(h, v : Integer);
(2) procedure SetClip(rgn : RgnHandle);
(2) procedure GetClip(rgn : RgnHandle);
(1) procedure ClipRect(r : Rect);
(1) procedure BackPat(pat : Pattern);

Pictures

(Picture Routines)

(2) function OpenPicture(picFrame : Rect) : PicHandle;
(2) procedure ClosePicture;
(2) procedure DrawPicture(myPicture : PicHandle; dstRect : Rect);
(2) procedure PicComment(kind, dataSize : Integer; dataHandle : QDHandle);
(2) procedure KillPicture(myPicture : PicHandle);

(Oval Routines)

(1) procedure FrameOval (r : Rect);
(1) procedure PaintOval (r : Rect);
(1) procedure EraseOval (r : Rect);
(1) procedure InvertOval (r : Rect);
(1) procedure FillOval (r : Rect; pat : Pattern);

(Graphical Operations on Regions)

(2) procedure FrameRgn (rgn) : RegHandle);
(2) procedure PaintRgn (rgn) : RgnHandle);
(2) procedure EraseRgn (rgn) : RgnHandle);
(2) procedure InvertRgn (rgn) : RgnHandle);
(2) procedure FillRgn (rgn) : RgnHandle; pat : Pattern);

Polygons

(Polygon Routines)

(2) function OpenPoly : PolyHandle;
(2) procedure ClosePoly;
(2) procedure KillPoly (poly : PolyHandle);
(2) procedure OffsetPoly (poly : PolyHandle; dh, dv : Integer);
(2) procedure MapPoly (poly : PolyHandle; fromRect, toRect : Rect);
(2) procedure FramePoly (poly : PolyHandle);
(2) procedure PaintPoly (poly : PolyHandle);
(2) procedure ErasePoly (poly : PolyHandle);
(2) procedure InvertPoly (poly : PolyHandle);
(2) procedure FillPoly (poly : PolyHandle; pat : Pattern);

```
(1)        notSrcOr       = 5;
(1)        notSrcXor      = 6;
(1)        notSrcBic      = 7;
(1)        patCopy        = 8;
(1)        patOr          = 9;
(1)        patXor         = 10;
(1)        patBic         = 11;
(1)        notPatCopy     = 12;
(1)        notPatOr       = 13;
(1)        notPatXor      = 14;
(1)        notPatBic      = 15;

    ( QuickDraw color separation constants )

(2)        normalBit      = 0;    { normal screen mapping }
(2)        InversBit      = 1;    { inverse screen mapping }
(2)        redBit         = 4;    { RGB additive mapping }
(2)        greenBit       = 3;
(2)        blueBit        = 2;
(2)        cyanBit        = 8;    { CMYBk subtractive
(2)        magentaBit     = 7;       mapping }
(2)        yellowBit      = 6;
(2)        blackBit       = 5;
(2)        blackColor     = 33;   { colors expressed in
(2)        whiteColor     = 30;      these mappings }
(2)        redColor       = 205;
(2)        greenColor     = 341;
(2)        blueColor      = 409;
(2)        cyanColor      = 273;
(2)        magentaColor   = 137;
(2)        yellowColor    = 69;

    ( standard picture comments )

(2)      picLParen  = 0;
(2)      picRParen  = 1;
(1)   type QDByte   = -128..127;
(1)      QDPtr      = ^QDByte;     { blind pointer }
(1)      QDHandle   = ^QDPtr;      { blind handle }
(1)      Str255     = string[255];
(1)      Pattern    = packed array [0..7] of 0..255;
(1)      Bits16     = array [0..15] of Integer;
(1)      VHSelect   = (v, h);
(2)      GrafVerb   = (frame, paint, erase, invert, full);
(1)      StyleItem  = (bold, italic, underline, outline,
                        shadow, condense, extend);
(1)      Style      = set of StyleItem;
(1)      FontInfo   = record
                        ascent  : Integer
                        descent : Integer
                        WidMax  : Integer
                        leading : Integer
                      end;
```

The comments in the left-hand margin indicate whether the declared item is to be found in QuickDraw1 (1) or QuickDraw2 (2).

```
(1)   Rect = record case Interger of
             0: (top       : Integer;
                 left      : Integer;
                 bottom    : Integer;
                 right     : Integer );
             1: (topLeft   : Point;
                 botRight  : Point );
            end;
(1)   BitMap = record
                 baseAddr  : QDPtr;
                 rowBytes  : Integer;
                 bounds    : Rect;
               end;
(1)   Cursor = record
                 data      : Bits16;
                 mask      : Bits16;
                 hotSpot   : Point;
               end;
(1)   PenState = record
                 pnLoc     : Point;
                 pnSize    : Point;
                 pnMode    : Integer;
                 pnPat     : Pattern;
               end
(2)   PolyHandle  = ^PolyPtr;
(2)   PolyPtr     = ^Polygon;
(2)   Polygon     = record
                      polySize   : Integer;
                      polyBBox   : Rect;
                      polyPoints : array [0..0] of Point;
                    end;
(2)   RgnHandle   = ^RgnPtr;
(2)   RgnPtr      = ^Region;
(2)   Region      = record
                      rgnSize    : Integer;     { rgnSize = 10 for
                                                   rectangular }
                      rgnBBox    : Rect;        { plus more data if not
                                                   rectangular }
                    end;
(2)   PicHandle   = ^PicPtr;
(2)   PicPtr      = ^Picture;
(2)   Picture     = record
                      picSize    : Integer;
                      picFrame   : Rect;        { plus byte codes for
                    end;                           picture content }
```

```
                  rRectProc    : QDPtr;
                  ovalProc     : QDPtr;
                  arcProc      : QDPtr;
                  polyProc     : QDPtr;
                  rgnProc      : QDPtr;
                  bitsProc     : QDPtr;
                  commentProc  : QDPtr;
                  txMeasProc   : QDPtr;
                  getPicProc   : QDPtr;
                  putPicProc   : QDPtr;
                end;
(2)   GrafPtr   = ^GrafPort;
(2)   GrafPort  = record
                   device      : Integer;
                   portBits    : BitMap;
                   portRect    : Rect;
                   vsRgn       : RgnHandle;
                   clipRgn     : RgnHandle;
                   bkPat       : Pattern;
                   fillPat     : Pattern;
                   pnLoc       : Point;
                   pnSize      : Point;
                   pnMode      : Integer;
                   pnPat       : Pattern;
                   pnVis       : Integer;
                   txFont      : Integer;
                   txFace      : Style;
                   txMode      : Integer;
                   txSize      : Integer;
                   spExtra     : LongInt;
                   fgColor     : LongInt;
                   bkColor     : LongInt;
                   colrBit     : Integer;
                   patStretch  : Integer;
                   picSave     : QDHandle;
                   rgnSave     : QDHandle;
                   polySave    : QDHandle;
                   grafProcs   : QDProcsPtr;
                 end
(2)   var thePort       : GrafPtr;
(1)       white         : Pattern;
(1)       black         : Pattern;
(1)       gray          : Pattern;
(1)       ltGray        : Pattern;
(1)       dkGray        : Pattern;
(1)       arrow         : Cursor;
(1)       screenBits    : BitMap;
(1)       randSeed      : LongInt;
```

(Region Calculations)

```
(2)   function NewRgn : RgnHandle;
(2)   procedure DisposeRgn(rgn : RgnHandle);
(2)   procedure CopyRgn(srcRgn, dstRgn : RgnHandle);
(2)   procedure SetEmptyRgn(rgn : RgnHandle);
(2)   procedure SetRecRgn(rgn : RgnHandle;
                    left, top, right, bottom : Integer);
(2)   procedure RectRgn(rgn : RgnHandle; r : Rect);
(2)   procedure OpenRgn;
(2)   procedure CloseRgn(dstRgn : RgnHandle);
(2)   procedure OffsetRgn(rgn : RgnHandle; dh, dv : Integer);
(2)   procedure MapRgn(rgn : RgnHandle;fromRect, toRect : Rect);
(2)   procedure InsetRgn(rgn : RgnHandle; dh, dv : Integer);
(2)   procedure SectRgn(srcRgnA, srcRgnB, dstRgn : RgnHandle);
(2)   procedure UnionRgn(srcRgnA, srcRgnB, dstRgn : RgnHandle);
(2)   procedure DiffRgn(srcRgnA, srcRgnB, dstRgn : RgnHandle);
(2)   procedure XorRgn(srcRgnA, srcRgnB, dstRgn : RgnHandle);
(2)   function EqualRgn(rgnA, rgnB : RgnHandle) : boolean;
(2)   function EmptyRgn(rgn : RgnHandle) : boolean;
(2)   function PtInRgn(pt : Point; rgn : RgnHandle) : boolean;
(2)   function RectInRgn(r : Rect; rgn : RgnHandle) : boolean;
```

(Graphical Operations on Rectangles)

```
(1)   procedure FrameRect (r : Rect);
(1)   procedure PaintRect (r : Rect);
(1)   procedure EraseRect (r : Rect);
(1)   procedure IntertRect (r : Rect);
(1)   procedure FillRect (r : Rect; pat : Pattern);
```

(Rectangle Calculations)

```
(1)   procedure SetRect(var r : Rect; left, top, right, bottom : Integer);
(1)   function EqualRect (rect1, rect2 : Rect) : boolean;
(1)   function EmptyRect(r : Rect) : boolean;
(1)   procedure OffsetRect(var r : Rect; dh, dv : Integer);
(1)   procedure MapRect(var r : Rect; fromRect, toRect : Rect);
(1)   procedure InsetRect(var r : Rect; dh, dv : Integer);
(1)   function SectRect (src1, src2 : Rect; var dstRect : Rect) : boolean;
(1)   procedure UnionRect(src1, src2 : Rect; var dstRect : Rect);
(1)   function PtInRect(pt : Point; r : Rect) : boolean;
(1)   procedure Pt2Rect(pt1, pt2 : Point; var dstRect : Rect);
```

(Arc Routines)

```
(1)   procedure FrameArc (r : Rect; startAngle, arcAngle : Integer);
(1)   procedure PaintArc (r : startAngle, arcAngle : Integer);
(1)   procedure EraseArc (r : startAngle, arcAngle : Integer);
(1)   procedure InvertArc (r : startAngle, arcAngle : Integer);
(1)   procedure FillArc (r : rect; startAngle, arcAngle : Integer; pat : Pattern);
(1)   procedure PtToAngle (r : Rect; pt : Point, var angle : Integer);
```

(RoundRect Routines)

```
(1)   procedure FrameRoundRect (r : Rect; ovWd, ovHt : Integer);
(1)   procedure PaintRoundRect (r : Rect; ovWd, ovHt : Integer);
(1)   procedure EraseRoundRect (r : Rect; ovWd, ovHt : Integer);
(1)   procedure InvertRoundRect (r : Rect; ovWd, ovHt : Integer);
(1)   procedure FillRoundRect (r : Rect; ovWd, ovHt : Integer; pat : Pattern);
```

Common Procedures

```
Frame      (...);
Paint      (...);
Erase      (...);
Invert     (...);
Fill       (...);
```

```
      { Point Calculations }
(1)   procedure AddPt(src : Point; var dst : Point);
(1)   procedure SubPt(src : Point; var dst : Point);
(1)   procedure SetPt(var pt : Point, h, v : Integer);
(1)   function EqualPt(pt1, Pt2 : Point) : boolean;
(1)   procedure ScalePt(var pt : Point: from Rect, to Rect : Rect);
(1)   procedure MapPt (var pt : Point; fromRect, toRect : Rect);
(1)   procedure LocalToGlobal(var pt : Point);
(1)   procedure GlobalToLocal(var pt : Point);

      { Line Routines }
(1)   procedure HidePen;
(1)   procedure ShowPen;
(1)   procedure GetPen(var pt : Point);
(1)   procedure GetPenState(var pnState : PenState);
(1)   procedure SetPenState(    pnState : PenState);
(1)   procedure PenSize(width, height : Integer);
(1)   procedure PenMode(mode : Integer);
(1)   procedure PenPat(pat : Pattern);
(1)   procedure PenNormal;
(1)   procedure MoveTo( h,  v : Integer);
(1)   procedure Move   (dh, dv : Integer);
(1)   procedure LineTo( h,  v : Integer);
(1)   procedure Line   (dh, dv : Integer);

      { Text Routines }
(1)   procedure TextFont(font : Integer);
(1)   procedure TextFact(face : Style);
(1)   procedure TextMode(mode : Integer);
(1)   procedure TextSize(size : Integer);
(1)   procedure SpaceExtra(extra : LongInt);
(1)   procedure DrawChar(ch : Char);
(1)   procedure DrawString (s : Str255);
(1)   procedure DrawText(textBuf : QDPtr;
                         firstByte, byteCount : Integer);
(1)   function CharWidth(ch : Char); : Integer;
(1)   function StringWidth(s : Str255); : Integer;
(1)   function TextWidth(textBuf : QDPtr;
                         firstByte, byteCount : Integer) : Integer;
(1)   procedure GetFontInfo(var info : FontInfo);

      {Bottleneck Interface }
(2)   procedure SetStoProcs(var procs : QDProcs);

(2)   procedure StdText(count : Integer;
                       textAddr : QDPtr; numer, denom : Point);
(2)   procedure StdLine(newPt : Point);
(2)   procedure StdRect (verb : GrafVerb; r : Rect);
(2)   procedure StdRect (verb : GrafVerb; r : Rect; ovWd, ovHt : Integer);
(2)   procedure StdOval (verb : GrafVerb; r : Rect);
(2)   procedure StdArc (verb : GrafVerb; r : Rect;
                       startAngle, arcAngle : Integer);
(2)   procedure StdPoly(verb : GrafVerb; poly : PolyHandle);
(2)   procedure StdRgn (verb : GrafVerb; rgn : RgnHandle);
(2)   procedure StdBits(var srcBits : BitMap;
                       var srcRect, dstRect : Rect;
                           mode : Integer;
                           maskRgn : RgnHandle);
(2)   procedure StdComment(kind, dataSize : Integer; dataHandle : QDHandle);
(2)   function StdTxMeas(    count : Integer;
                             textAddr : QDPtr;
                         var numer, denom : Point;
                         var info : FontInfo) : Integer;
(2)   procedure StdGetPic(dataPtr : QDPtr; byteCount : Integer);
(2)   procedure StdPutPic(dataPtr : QDPtr; byteCount : Integer);

      { Graphical Operations on BitMaps }
      procedure ScrollRect(dstRect : Rect;
                           dh, dv : Integer; updateRgn : rgnHandle);
      procedure CopyBits (srcBits, dstBits : BitMap;
                          srcRect, dstRect : Rect;
                          mode : Integer;
                          maskRgn : RgnHandle);

      { Cursor Routines }
(1)   procedure InitCursor;
(1)   procedure SetCursor(crse : Cursor);
(1)   procedure HideCursor;
(1)   procedure ShowCursor;
(1)   procedure ObscureCursor;

      { Misc Utility Routines }
(1)   function Get Pixel(h, v : Integer) : boolean;
(1)   function Randon : Integer;
(1)   procedure StuffHex(thingptr : QDPtr; s : Str255);
(2)   procedure ForeColor(color : LongInt);
(2)   procedure BackColor(color : LongInt);
(2)   procedure BackBit (whichBit : Integer);
```

```
(1)  const srcCopy      = 0;    { the 16 transfer modes }
(1)        srcOr        = 1;
(1)        srcXor       = 2;
(1)        srcBic       = 3;
(1)        notSrcCopy   = 4;
```

```
(1)  Point = record case Integer of;
           0: (v  = Integer;
               h  = Integer );
           1: (vh = array [VHSelect] of Integer);
           end;
```

```
(2)  QDProcsPtr = ^QDProcs;
(2)  QDProcs    = record
                    textProc    : QDPtr;
                    lineProc    : QDPtr;
```

STRUCTURED TYPES

RECORD

Definition :

```
TYPE
    Typename = RECORD
      field : fieldtype;
         .
         .
         .
      field : fieldtype
    END;
```

Example:

```
TYPE
    Class =RECORD
      Max : Integer;
      Size : Integer;
      Full : Boolean
    END;
    VAR
      Math : Class;
    Math.Size := 25;
    Math.full := FALSE;
```

VARIANT RECORD

Definition :

```
TYPE
    Typename = RECORD
      CASE fieldname
        fieldtype OF
        const : fields;
            .
            .
            .
        const : fields;
    END;
```

Example:

```
TYPE
    Class =RECORD
      Max : Integer;
      Size : Integer;
      Case Full : Boolean
       OF
        True : (EmptySeats
         : Integer);
        Fasle : ( )
    END;
```

FILE

Definition :

```
TYPE
    Typename = FILE OF
      any type:
    or
    TYPE
        TEXT = FILE OF
        CHAR;
```

Example:

```
TYPE
    TEXT =FILE OF CHAR;
    VAR
      Class : TEXT;
    Writeln (Class, 'Pro-
    gramming!');
```

POINTER

Definition :

```
TYPE
    pointername =
      ↑anytype;
```

Example:

```
TYPE
    listptr =↑list;
      list = RECORD
        item : integer;
        next : listptr
    END;
```

PROGRAM STRUCTURE

```
PROGRAM     program name (INPUT, OUTPUT);
   LABEL    label #, . . . , label #;
   CONST    constname = value;
                .
                .
                .
            constname = value;
   TYPE     typename = definition;
                .
                .
                .
            typename = definition;
   VAR      varname : vartype;
                .
                .
                .
            varname : vartype;
  'ROCEDURE or FUNCTION
BEGIN
   Statements
END.
```

CONTROL STRUCTURES

```
IF condition
THEN statement
ELSE statement
            The ELSE portion is optional and can be
            omitted.
FOR         varname := bound TO (or DOWNTO)
             bound
DO          statement;
WHILE       condition
DO          statement;
REPEAT      statement;
                .
                .
                .
            statement
UNTIL       condition;
CASE        varname OF value : action
                .
                .
                .
            value : action
END;
            Every possible value for the type of
            varname must be listed.
```

MISCELLANEOUS

```
GOTO        lable #;
WITH .. DO  recordname
            statement involving the field(s) of
            recordname;
```

SCALAR

USER-DEFINED

Definition :

```
TYPE
    Typename = (Const,
      ....Const);
```

Example:

```
TYPE
    Fruit =(Apple
      Orange, Pear);
    VAR
      Snack : Fruit;
      Snack : Orange;
```

SUBRANGE

Definition :

```
TYPE
    Typename = Const..
      Const;
     Const can be any
     scalar type
     except REAL.
```

Example:

```
TYPE
    Day = (M,T,W,R,F,
      Sa,Su);
    Workday = M..F;
```

STRUCTURED

SET

Definition :

```
TYPE
    Typename = SET
      OF scalar type;
```

Example:

```
TYPE
    Day = (M,T,W,R,F,
      Sa,Su);
    DaySet = SET OF
      Day;
```

ARRAY

Definition :

```
TYPE
    Typename = ARRAY
      [bound..bound,
        ...bound..bound]
        Of any type;
```

Example:

```
TYPE
    Day = (M,T,W,R,F,
      Sa,Su);
    Schedule = ARRAY
      [M..F] OF Day;
    Age = ARRAY [1..
      120] OF INTEGER;
```

FUNCTIONS

ARITHMETIC

		TYPES I	 R
ABS	Y := ABS(X), Y becomes $|X|$.	X Y	X Y
COS	Y := COS(X), Y becomes cos(X), X is in radians.	X	X Y
SIN	Y := SIN(X). Y becomes sin(X).	X	X Y
ARCTAN	Y := ARCTAN(X), Y becomes arctan(X).	X	X Y
EXP	Y := EXP(X), Y becomes e^X.	X	X Y
LN	Y := LN(X), Y becomes $\ln_e x$. X must be >0.	X	X Y
SQR	Y := SQR(X), Y becomes x^2.	X Y	X Y
SQRT	Y := SQRT(X), Y becomes $\sqrt{X}$.	X	X Y
ROUND	Y := ROUND(X), Y becomes the integer nearest X.	Y	X
TRUNC	Y := TRUNC(X), Y becomes the integer between X and O nearest X.	Y	X

ORDERING

		TYPES C	 I
PRED	Y := PRED(X), Y becomes the character preceeding X in the character set.	X Y	
SUCC	Y := SUCC(X), Y becomes the character succeeding X in the character set.	X Y	
ORD	Y := ORD(X), Y becomes the position number of X in character set.	X	Y
CHR	Y := CHR(X), Y becomes the character at position on number X in the character set.	Y	X

BOOLEAN

		TYPES B	 I
ODD	Y := ODD(X), Y becomes true if X is odd else Y becomes false.	Y	X
EDLN	EDLN, True if next character is the end-of-the-line control character.		
EDF	EDF, True if next character is the end-of-file control character.		

TYPES KEY: B—boolean, C—Char, I—Integer, R—Real

PROCEDURES

INPUT/OUTPUT

		TYPES C	 F	 I	 R
READ	READ(X), Reads a value from the input file	X	X	X	X
READLN	READLN(X), Reads a value from the input file and advances to the next input line	X	X	X	X
WRITE	WRITE(X), Writes a value on the output line	X	X	X	X
WRITELN	WRITELN(X), Writes a value on the output line and advances to the next output line	X	X	X	X

TRANSFER

UNPACK	UNPACK(X,Y,Z), The contents of packed array X are assigned to the not packed array Y starting at index integer Z.
PACK	PACK(X,Y,Z), The contents of not packed array X are assigned to the packed array Z starting at index integer Y.

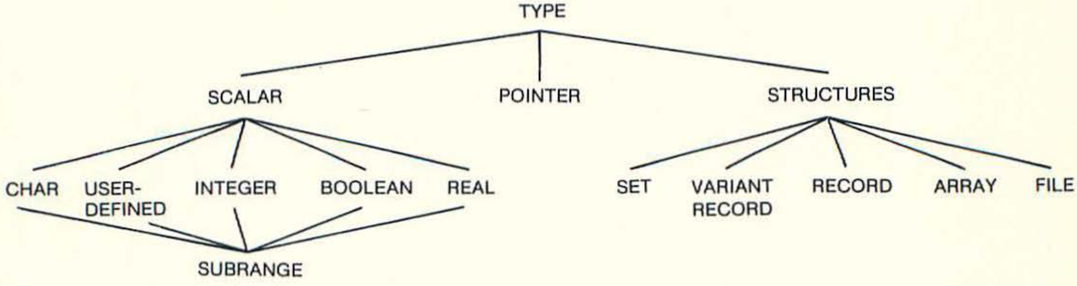
FILE HANDLING

RESET	RESET(X), Sets the file X pointer to the beginning of file X.
REWRITE	REWRITE(X), Creates an empty file X. Erases any file X contents.
GET	GET(X), Moves the file X pointer to the next character in file X.
PUT	PUT(X), Places the value of the file X pointer at the end of file X.

DYNAMIC ALLOCATION

NEW	NEW(X), Refers pointer X to a memory location.
DISPOSE	DISPOSE(X), Frees the memory location that X refers to.

TYPES KEY: C—Char, F—File, I—Integer, R—Real



OPERATORS

	EXAMPLE	B	C	I	P	R	S	U
:=	a := b, a becomes b	a b	a b	a b	a b	a b		a b
*	a := b * c, If a, b, and c are sets : a becomes b∩c, else a becomes b · c.			a b c		a b c	a b c	
/	a := b/c, a becomes b/c.			b c		a b c		
+	a := b + c, If a, b, and c are sets : a becomes b∪c, else a becomes b + c.			a b c		a b c	a b c	
-	a := b - c, If a, b, and c are sets : a becomes b - c (b∩c), else a becomes b - c.			a b c		a b c	a b c	
<	a < b , true if a preceeds b in the type definition else false.	a b	a b	a b		a b		a b
>	a > b , true if a succeeds b, else false.	a b	a b	a b		a b		a b
=	a = b , true if a is the same as b.	a b	a b	a b		a b		a b

	EXAMPLE	B	C	I	P	R	S	U
< =	a < = b, true if a precedes b or a is the same as b	a b	a b	a b		a b		a b
> =	a > = b, true if a succeeds b or a is the same as b	a b	a b	a b		a b		a b
<>	a <> b, true if a is not the same as b	a b	a b	a b		a b		a b
DIV	a := b DIV c, a becomes b÷c - the remainder			a b c				
MOD	a := b MOD c, a becomes the remainder of b ÷ c			a b c				
NOT	NOT a, true if a is false and false if a is true	a						
AND	a AND b, true only if a is true and be is true, else false.	a b						
OR	a OR b, true if a is true, b is true, or both a and b are true.	a b						
IN	a IN b, true if a is an element of set b	a	a	a		a	b	a

TYPES KEY: B—boolean, C—char, I—integer, P—pointer, R—real, S—set, U—user-defined

SCALAR TYPES

INTEGER

Implied Definition :

```
TYPE
   Integer = (-___,
     . . . ,+___);
```

Example:

```
VAR
   Total : INTEGER
Total := 14;
Total := 2 * 46;
```

CHAR

Implied Definition :

```
TYPE
   Char = (___, . . . ,___);Const
```

Example:

```
VAR
   Alpha : CHAR;
Alpha := 'S';
```

BOOLEAN

Implied Definition :

```
TYPE
   Boolean = (FALSE,
     TRUE);
```

Example:

```
VAR
   Same : BOOLEAN;
Same := TRUE;
```

REAL

Implied Definition :

```
TYPE
   Real = (-___E+___. .
     -___E-___,O.O,
     +___E-___. .+___E+___);
```

Example:

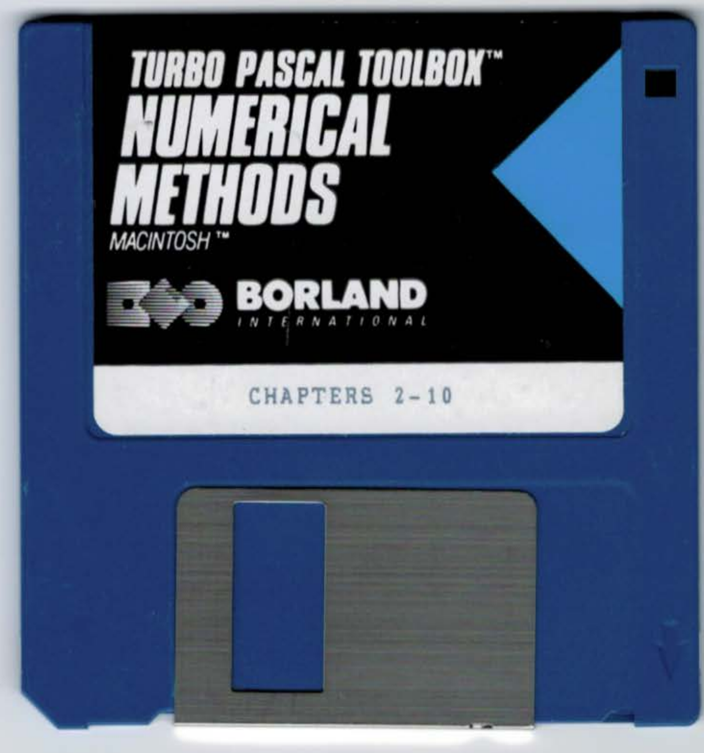
```
VAR
   Root : REAL;
   Hyp : REAL;
Root := SQRT(hyp);
```

SYNTAX SUMMARY

T. G. Lewis, Abbas Birjandi

From *Macintosh™ Hands-On Pascal* by T. G. Lewis and Abbas Birjandi. Wadsworth Publishing Company, Belmont, CA.

TURBO PASCAL TOOLBOX™
NUMERICAL METHODS
MACINTOSH™
BORLAND
INTERNATIONAL
DEMO DISK
X1A0035915


TURBO PASCAL TOOLBOX™
NUMERICAL METHODS
MACINTOSH™
BORLAND
INTERNATIONAL
CHAPTERS 2-10

Macintosh™ Hands-On **Pascal**

Inside—
The easy way to learning
***Pascal** on the Macintosh*

Pascal programming on the Macintosh becomes easy and enjoyable when you use Lewis and Birjandi's new primer. MACINTOSH™ HANDS-ON PASCAL is special because it creates an environment that encourages your participation from the start. In fact, each chapter is actually a session at the Macintosh where you'll learn Pascal concepts in close interaction with the machine. There are also plenty of detailed examples, interesting case studies, and excellent problem-solving sections in each chapter to reinforce what you've just learned. With this kind of hands-on experience and backup support, you'll soon be programming Pascal on the Macintosh like a seasoned pro.

Other features:

Covers the entire Pascal language, plus the Graphics and Sound features of the Macintosh

Includes a tear-out Pascal syntax chart and a tear-out Quickdraw reference card

Contains an appendix on debugging Pascal programs

Macintosh is a trademark licensed to Apple Computer, Inc.
Illustration by Stephen Jacobs

ISBN 0-534-06354-3