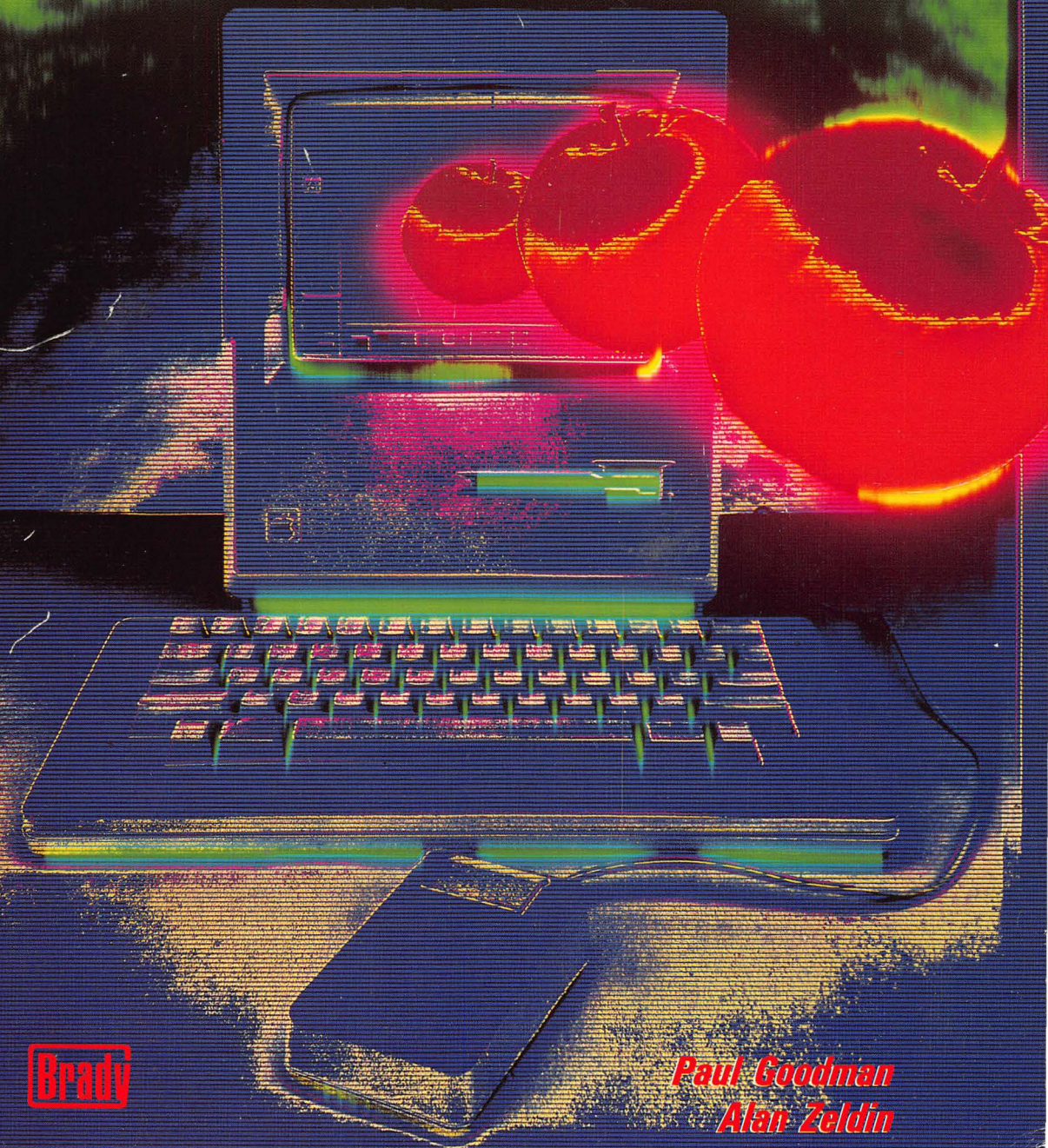


# *The MacPascal Book*



**Brady**

*Paul Goodman  
Alan Zeldin*



B

# The MacPascal Book



Publishing Director: David Culverwell  
Acquisitions Editor: Susan Love  
Production Editor/Text Design: Roberta Glencer  
Art Director/Cover Design: Don Sellers  
Assistant Art Director: Bernard Vervin  
Manufacturing Director: John Komsa

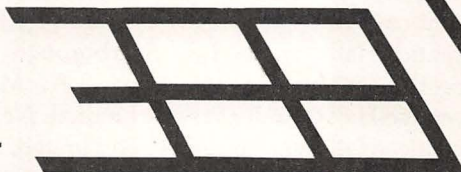
Typesetter: Shepard Poorman Communications, Indianapolis, IN  
Printer: R. R. Donnelley & Sons, Harrisonburg, VA  
Typefaces: Paladium (text), CG Frontiera (display), News Gothic (programs)



# **The MacPascal Book**

**Paul Goodman  
Alan Zeldin**

**Brady Communications Company, Inc.  
A Prentice-Hall Publishing Company  
Bowie, Maryland 20715**





## The MacPascal Book

Copyright © 1985 by Brady Communications Company, Inc. All rights reserved. No part of this publication may be reproduced or transmitted in any form or by any means, electronic or mechanical, including photocopying and recording, or by any information storage and retrieval system, without permission in writing from the publisher. For information, address Brady Communications Company, Inc., Bowie, Maryland 20715.

### Library of Congress Cataloging In Publication Data

Goodman, Paul.

The MacPascal book.

Bibliography: p.

Includes index.

1. Macintosh (Computer)—Programming. 2. PASCAL (Computer program language) I. Zeldin, Alan. II. Title.

QA76.8.M3G66 1985 001.64'2 85-431

ISBN 0-89303-644-7

Prentice-Hall of Australia, Pty., Ltd., *Sydney*

Prentice-Hall Canada, Inc., *Scarborough, Ontario*

Prentice-Hall Hispanoamericana, S.A., *Mexico*

Prentice-Hall of India Private Limited, *New Delhi*

Prentice-Hall International (UK) Limited, *London*

Prentice-Hall of Japan, Inc., *Tokyo*

Prentice-Hall of Southeast Asia Pte. Ltd., *Singapore*

Editora Prentice-Hall Do Brasil LTDA., *Rio de Janeiro*

Whitehall Books, Limited, *Petone, New Zealand*

Printed in the United States of America

85 86 87 88 89 90 91 92 93 94 95      1 2 3 4 5 6 7 8 9 10



# Contents

## **1 Computer Concepts / 1**

Hardware / 1

Software / 4

Language Translators / 4

Interpreters and Compilers / 5

The Operating System / 5

The Macintosh / 5

Macintosh Software / 6

## **2 Using MacPascal / 7**

Getting Up and Running / 7

Editing a Program / 13

Saving a Program / 14

Recalling a Program / 15

Printing a Program / 16

Printing the Active Window / 17

## **3 Pascal Fundamentals / 19**

Syntax / 19

Documenting a Program / 22

Write and Writeln / 23

Data / 26

Data Types / 26

Variables / 29

Assignment Statements / 31

More on Write and Writeln / 33

Expressions / 35

Operator Precedence / 39

Constants / 40

Read and Readln / 42

Review of Program Structure and Some Examples / 44

The Perimeter and Area of a Rectangle / 44

Converting Temperatures / 45

Exercises / 46

## **4 Pascal Structures / 47**

The Boolean Data Type / 47

Boolean Operators / 47

Boolean Expressions / 48

IF-THEN / 49

Compound Statements / 52

IF-THEN-ELSE / 53



|  |           |
|--|-----------|
| Nested IF Statements /                     | 55        |
| FOR Loops /                                | 56        |
| DOWNT0 /                                   | 61        |
| Calculating Interest Compounded Daily /    | 61        |
| The Summation of an Infinite Series /      | 63        |
| The WHILE Loop /                           | 64        |
| Sentinals /                                | 67        |
| The Break Even Point /                     | 68        |
| Controlling the Text Window /              | 68        |
| The Macintosh Screen /                     | 69        |
| Introduction to Graphics /                 | 71        |
| Exercises /                                | 74        |
| <br>                                       |           |
| <b>5 Debugging a MacPascal Program /</b>   | <b>77</b> |
| The Nature of a MacPascal Program /        | 77        |
| Execution Modes /                          | 78        |
| Syntax Errors /                            | 79        |
| Undeclared Identifiers /                   | 81        |
| Execution Errors /                         | 81        |
| Run Time Errors /                          | 81        |
| Logic Errors /                             | 81        |
| The Observe Window /                       | 82        |
| Setting Break Points /                     | 83        |
| The Instant Window /                       | 84        |
| <br>                                       |           |
| <b>6 More on Data Types /</b>              | <b>85</b> |
| The Char Types /                           | 85        |
| Ordinal Types /                            | 86        |
| The ORD and CHR Functions /                | 86        |
| The SUCC and PRED Functions /              | 88        |
| Other Built-In Functions /                 | 88        |
| The Conversion Functions—TRUNC and ROUND / | 88        |
| More on Reals and Integers /               | 89        |
| The Longint Data Type /                    | 89        |
| The Extended Real Types /                  | 90        |
| The Arithmetic Functions /                 | 91        |
| The Trigonometric Functions /              | 92        |
| The Logarithmic Functions /                | 93        |
| Tool Box Functions /                       | 93        |
| User-Defined Data Types /                  | 94        |
| Subranges /                                | 96        |
| Drawing Ovals /                            | 98        |
| Exercises /                                | 101       |



- 7 Procedures / 103**
  - Scope of Variables / 106
  - Parameter Passing / 109
    - Variable Parameters / 111
    - The Major Advantage / 113
    - Value Parameters / 113
  - Comparing Value and Variable Parameter Passing / 115
  - Mixing Variable and Value Parameters / 116
    - Mortgage Calculator / 117
  - Drawing Lines / 120
  - Exercises / 123
  
- 8 Arrays and Strings / 125**
  - Sentinels / 128
  - Two-Dimensional Arrays / 130
  - Arrays of Characters / 142
    - The Code Breaker Program / 144
  - Strings / 145
    - Reading a String / 146
    - Comparing Strings / 147
  - The String Functions and Procedures / 147
    - The Length Function / 147
    - The Concat Function / 147
    - The Pos Function / 148
    - The Copy Function / 148
    - The Delete Procedure / 148
    - The Omit Function / 148
    - The Insert Procedure / 149
    - The Include Function / 149
  - Exercises / 149
  
- 9 More On Structures / 151**
  - The Repeat Loop / 151
  - The Bubble Sort / 153
  - The Case Statement / 154
  - User-Defined Functions / 157
  - Recursion / 159
  - Records / 165
  - The With Statement / 168
  - Duplicate Field Names / 169
  - Arrays of Records / 170
  - Nested Records / 173
  - Time and Date Operations / 175
  - Sets / 176

|   |  |
|---|--|
| Set Operators / 177   |  |
| Set Union / 177   |  |
| Set Difference / 177  |  |
| Set Intersection / 178  |  |
| Exercises / 180   |  |
| <b>10 A Formal Look at Graphics / 185</b>                                 |  |
| Points / 185  |  |
| Drawing Lines / 186   |  |
| Rectangle / 187   |  |
| Controlling the Drawing Window / 189                                      |  |
| Drawing Rectangles / 190  |  |
| Other Shapes / 191  |  |
| The Pen / 192   |  |
| The Mouse / 195   |  |
| The Cursor / 197  |  |
| Sketchpad / 197   |  |
| A New Feature / 199   |  |
| Displaying Text on the Drawing Window / 199                               |  |
| Calculations with Rectangles / 202  |  |
| Implementing the New Feature / 203  |  |
| Playtime with Quick Draw / 205  |  |
| Exercises / 210   |  |
| <b>11 Files / 213</b>   |  |
| The File Buffer / 215   |  |
| Using Files / 215   |  |
| Opening Files / 216   |  |
| Accessing Files / 217   |  |
| Closing a File / 218  |  |
| Mixing Get and Put / 219  |  |
| Using Random Access Files / 219   |  |
| Seek / 220  |  |
| Finding the End of a File / 221   |  |
| Text Files / 221  |  |
| <b>12 Variant Records and Pointers / 233</b>                              |  |
| Variant Records / 233   |  |
| Pointers / 236  |  |
| <b>13 A Look at an Application—The Checking and Savings Program / 243</b> |  |
| Overview / 243  |  |
| Data Structures / 244   |  |
| Development / 245   |  |



**Appendices / 267**

A Selected Exercise Answers / 268

B Menu Summary / 270

C Documenting a Program / 275

D Sound and Music / 284

E Differences Between MacPascal and UCSD Pascal / 289

F MacPascal Reserved Words / 292

G MacPascal Syntax Diagrams / 293

H List of QuickDraw Routines / 308

I List of Sane Functions and Procedures / 314

J MacPascal Error Messages / 317

K Bibliography / 323

L The Macintosh Character Set / 325

**Index / 327**

## Limits of Liability and Disclaimer of Warranty

The author(s) and publisher of this book have used their best efforts in preparing this book and the programs contained in it. These efforts include the development, research, and testing of the theories and programs to determine their effectiveness. The author(s) and publisher make no warranty of any kind, expressed or implied, with regard to these programs or the documentation contained in this book. The author(s) and publisher shall not be liable in any event for incidental or consequential damages in connection with, or arising out of, the furnishing, performance, or use of these programs.

## Note to Authors

Have you written a book related to personal computers? Do you have an idea for developing such a project? If so, we would like to hear from you. Brady produces a complete range of books for the personal computer market. We invite you to write to David Culverwell, Publishing Director, Brady Communications Company, Inc., Bowie, MD 20715.

## Registered Trademarks

Macintosh is a trademark licensed by Apple Computer, Inc.  
MacWrite is a registered trademark of Apple Computer, Inc.



# Preface

On a cold day in the winter of 1642, in the northwest corner of France, a young man put the finishing touches on a strange device built from gears, pegs, and dials.

On a snowy day in 1971, in Zurich, Switzerland, a university professor put the finishing touches on a paper ready for publication.

On a sunny day in 1976, in a garage in northern California, two young men dressed in jeans put the finishing touches on a computer. No one could have predicted the resulting revolution.

History sometimes has ways of connecting events that take place hundreds of years and thousands of miles apart. None of these men could have realized the importance of his work in the future or completely understood his connection with the past.

On a hot day in 1984, in Lexington, Massachusetts, a team of computer scientists at Think Technologies, Inc., linked the works of Blaise Pascal—the 17th-century mathematician, natural philosopher, and inventor of the first adding machine—with Nikolas Wirth—designer of the Pascal computer language—and with Steven Jobs and Steven Wozniak—founders of Apple Computer. This team completed an extraordinary new computer language, Macintosh Pascal.

This is the legacy of this book. The MacPascal Book is intended for both the novice who wants to learn how to program and the experienced Pascal programmer who wants to use Macintosh Pascal. Embedded in this book is a full Pascal text that carefully teaches all the features of Pascal along with teaching programming and how to use the MacPascal system. A highlight of the book is its discussion of how to use “QuickDraw” graphics to produce animation and interesting graphics effects. Graphics are integrated into each chapter to demonstrate the topic and to provide some entertainment. A separate graphics chapter includes a video game program and ideas for others.

Macintosh Pascal is the easiest Pascal system to learn and use to date. Being a full implementation of Pascal, it allows a programmer to express problems in a natural form similar to how people think. Being Macintosh software, MacPascal makes full use of the unique features of the Macintosh, thus making MacPascal easy to learn, easy to use, and easy to debug the occasional programming mistakes that inevitably occur. This book contains all that you will need to become a proficient Pascal programmer.

Paul Goodman  
Alan Zeldin

# Introduction

This book teaches Pascal programming with the MacPascal system for the Macintosh computer. It is intended as a text for an introductory Computer Science course or as a self-study guide for Pascal.

In the past, Pascal books have been one of two types. There were books that were implementation free that taught Pascal in a vacuum, as if it never had to be run on a computer. These books were weak in dealing with I/O and file usage, which vary greatly from computer to computer.

The other books were primers on how to use Pascal on a specific system and gave only a light treatment to the Pascal language itself.

This book suffers from neither of these two pitfalls. It is a full Pascal text covering the entire breadth of the Pascal language in substantial detail and gives full coverage to the use of MacPascal, with detailed descriptions of file usage and I/O. A good example is how the book covers QuickDraw, the Macintosh's powerful graphics package. Most of the book's chapters include graphics-oriented examples to provide motivation to the reader; however, they are only used to support the concepts introduced in the chapter, not in lieu of traditional Pascal examples. In addition, this material is located in the back of a chapter, rather than in the middle, to make it easier for an instructor to omit if desired. For review and practice, each chapter includes sample problems based on the material in the chapter.

## Pedagogy and Organization

The authors of this book have taught over 3000 university students to program. The pedagogy of this book reflects their experiences in teaching Pascal. Chapters are not grouped by categories such as loops or control structures as found in many books, but rather Pascal structures are grouped by function, thus, quickly giving the reader a sufficient Pascal vocabulary to program with. This method avoids the frustration of having to wait to get to the middle of the book before the reader can start programming.

Chapter 1 of the book is a general introduction to computer hardware and software concepts as they appear on the Macintosh.

Chapter 2 introduces the reader to the MacPascal language and environment with step-by-step descriptions of how to enter, edit, and run a simple program. This chapter quickly motivates and prepares the reader for Chapter 3, where Pascal fundamentals are introduced. Covered are data, data types, variables, assignment, expressions, program structure, input/output, syntax diagrams, and documentation. Early exposure is



provided to structured design and programming concepts in several simple programming examples.

Chapter 4 covers the Pascal structures needed to start implementing more substantial programs. Decision making with the IF statement and the FOR and WHILE loops is discussed. Numerous examples with different data types are used to illustrate these techniques. Chapter 4 also introduces the Macintosh graphics coordinate system along with simple graphics and animation programs.

Chapter 5 takes a full look at the MacPascal programming environment and discusses errors and debugging. The powerful MacPascal debugging tools are demonstrated.

Chapter 6 takes an in-depth look at data types. The MacPascal extended real and integer types are discussed as well as built-in functions. Included with the built-in functions are several Macintosh Toolbox functions used to manipulate the Macintosh's environment. This chapter concludes with several new graphics commands and an animation program using built-in functions.

Traditionally, the hardest Pascal concept to grasp is parameter passing. Chapter 7 emphasizes this concept where procedures are covered. Functions are covered in a later chapter because experience has taught that introducing both procedures and functions at the same time tends to lead to confusion. Simple to follow examples are used to drive home the concepts involved.

Chapter 8 covers arrays and strings and Chapter 9 covers the remaining structures: functions, records, the CASE statement, and sets. A highlight of the book is its coverage of recursion. Several simple, yet fascinating graphics programs physically demonstrate this powerful, yet intimidating programming technique.

A more formal presentation of graphics is the goal of Chapter 10. Two complete interactive graphics systems making full use of the graphics capability and the mouse are developed. These programs serve as a model for implementing interactive graphics programs.

Chapter 11 covers files and file processing in the MacPascal implementation. Both sequential and random file access are demonstrated.

The two remaining Pascal topics, variant records and pointers, are covered in Chapter 12. The book's final chapter develops a programming system with file processing. The same top-down design techniques emphasized throughout the book are utilized.

The book contains appendices covering selected answers to problems, creating sound, and associated music theory, the Standard Apple Numeric Environment, syntax diagrams, and other reference tables.

MacPascal is the most effective language available for teaching programming. Since it is interpreted rather than compiled, it provides unique debugging facilities not found in any other Pascal implementation process. The illuminating Observe window allows the programmer

to see the value of variables while stepping through a program line by line. This development tool turns program bugs into a learning opportunity. Furthermore, since it is interpreted, the programmer avoids the time-consuming compiling and linking process every time a slight program change is made. This fosters experimentation by the programmer, which leads to improved learning and understanding.

## Special Features of the Book

**Complete Pascal Coverage.** The text includes the topics neglected by similar books; sets, variant records, pointers, units, sound and SANE (the Apple Standard Numeric Environment) are all covered.

**Emphasis on Program Development.** Starting in the first chapter, program development is stressed by encouraging top-down programming design and by including pseudocode of most programs.

**Total Coverage of Files.** The use of files occupies two complete chapters rather than the five or six pages usually devoted to this important topic. The reader is presented with the necessary information and programming techniques to develop programming systems to solve substantial data storage applications.

**In Depth Look at Graphics.** A majority of the Macintosh QuickDraw routines is presented. A difficult subject to grasp, QuickDraw concepts are first introduced in Chapter Four. The concepts are built upon from chapter to chapter. A separate chapter is devoted to a formal description of QuickDraw and includes two interactive graphics programs, SketchPad, a free-hand drawing program, and Paddle-Ball, a video game. The reader is also presented with the necessary information to develop his own graphic interfaces.

**The ToolBox.** The text covers the MacPascal routines that access the Macintosh User Interface ToolBox. Demonstrated are ways to utilize the Mouse, the built-in clock, to generate sounds, and many other useful routines.

**Sound and Music.** A concise Appendix covers music theory and how to play songs with the Note procedure.

**Complete Programming Systems Presented.** The Book progresses from simple programs to complete menu-driven programming systems utilizing files. The book's final chapter develops from scratch a multileveled, menu-driven bank account tracking program. This program serves as a model for other complete systems.



**Appendices.** The MacPascal book contains appendices that complement the text with easy-to-reference information on menu commands, documenting programs, sound, UCSD Pascal, reserved words, syntax diagrams, QuickDraw procedures, SANE procedures, error messages, bibliography, and the Macintosh character set.

## About the Authors

PAUL GOODMAN is an Instructor of Computer Science at Queens College of the City University of New York. Goodman was one of the first in the country to teach Pascal. He is also the author of *The Commodore 64 Guide to Data Files and Advanced Basic* (Brady Communications Company).

ALAN ZELDIN is a Software Engineer at Core Systems Solutions in New York City and an Adjunct Instructor of Computer Science at Queens College of the City University of New York. He is also developer of SPY'S DEMISE, a popular arcade action game for the Apple II, and other 6502 computer programs published by Penguin Software.



# Acknowledgments

On a cool spring night in a hi-rise apartment in New York City, two writers commenced work on a Pascal text book. Slowly but surely, many late nights of work became a book. It was not without the help of many people that this transformation took place.

No book is published without hard work and dedication by the people who form a publishing company. We wish to thank the entire staff at Brady Communications but especially David C. Culverwell, Sue Love, Christi Mangold, Bobbie Glencer, John Yarley, and the Art department.

Nor is a book published without the contributions made by prepublication reviewers. Susan Schmieman, Bill O'Brien, Peter Leeds, Rico Vaccaro, Richard Phillips, and our colleague Ken Lord, all provided insightful criticism and suggestions that were of great help to us.

Special thanks go to Charles Weg of Apple Computer for the information and encouragement he provided.

Finally, just as a book is shaped by many people, so are writers. Over the years there have been friends and family, teachers and students, co-workers and employers, who have helped form the way we think about our work and about life itself. This book is dedicated to them.

# 1 Computer Concepts

During the last several years computers have become an important force in our lives. With the advent of small, powerful, and inexpensive computers such as the Macintosh, the realm of knowledge and information processing, which was once available exclusively to large corporations and government, is now available on our desk tops. In order to make the best use of these powerful machines, it is necessary for us to have some understanding of how computers work and how to make computers work for us. In this chapter we will look at the major components that make up a computer system.

Computer systems have two main components, Hardware and Software. The hardware is the actual machine itself, the Macintosh for example. The software is the instructions that direct the hardware to do useful computations. A good analogy might be a high fidelity music system. The receiver, speakers, and turntable are hardware, and the music is the software.

## Hardware

Hardware can be subdivided further into several components:

**Input/Output Devices.** These devices transfer information from inside the computer to the outside world (output) or from the outside world into the computer (input). Examples of output devices on the Macintosh are the display screen, which "outputs" text and pictures, and the speaker, which "outputs" beeps and music. Examples of input devices on the Macintosh include the keyboard, which is used for entering text and numeric information, and the mouse, which is useful for making choices and "inputting" spacial information.



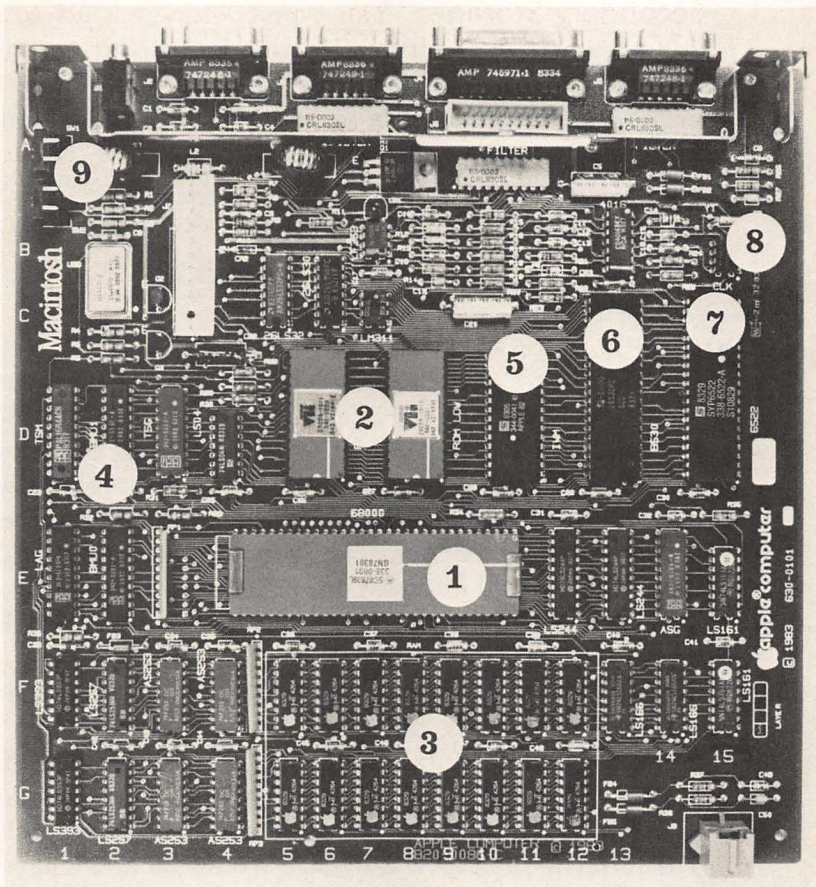


Figure 1-1.

**Central Processing Unit.** The CPU is the part of the computer that does the computation (for example, arithmetic) and logical (for example, decision making) operations. The CPU is sometimes called the “brain” of the computer; it is the component that interprets the programs and directs other parts of the computer to take appropriate action. The CPU inside the Macintosh is the Motorola 68000.

**Primary Storage.** Also referred to as Main Memory or just memory. Primary Storage is the place where programs and data are stored. Memory can store many different types of information including instructions, numbers, pictures, and text characters.

A byte is the amount of memory required to store one character. A “K” is equal to  $2^{10}$  or 1024 and is a standard computer abbreviation. Since 1024 is close to a thousand, the term “K” which stands for kilo (the Greek term for thousand) is used. The standard Mac is said to have 128K bytes of memory. This means that it can store just over 130,000 text characters. The 512K Mac-



- |                                  |  |
|----------------------------------|--|
| 1. Motorola 68000 Microprocessor | 6. 8530 Serial Communications Controller |
| 2. ROM                           | 7. 6522 Versatile Interface Adapter      |
| 3. RAM                           | 8. Real Time Clock                       |
| 4. Business Management Unit      | 9. Audio Connector                       |
| 5. Incredible WOZ Machine        |  |

Figure 1-2.

intosh (the Fat Mac) can store just over 534,000 text characters.

Memory is used to store information that needs to be accessed quickly, such as the instructions in the program that are currently running on the computer or data that the currently running program needs to access often. The Macintosh also has an additional 64K of Read Only Memory (ROM) which permanently stores the Toolbox. The Toolbox is what gives the computer its unique capabilities.



**Secondary Storage.** This type of storage is use to store data and programs, but information that is stored in secondary storage cannot be accessed as quickly as information that is in main memory. An example of a secondary storage device on the Macintosh is the micro-floppy disk drive. Programs that are not currently running on the machine can be stored on a floppy disk and loaded into memory only when they are to be run.

## Software

A program is a sequence of statements that instructs the computer to perform a specific task. Computer programs are written in languages that are designed to communicate very precisely and accurately the intentions of the programmer to the machine. The machine stores all information, programs, data, text and pictures internally as numbers. These numbers that make up the instructions of a computer program are called machine language. Because it is difficult for people to understand machine language, computer scientists have developed programming languages much closer to natural languages, such as English. These programming languages are called high level languages, as opposed to machine language, which is referred to as a low level language. The ultimate high level language would be a language like English. There are no machines that can understand English, therefore programmers use high level languages, such as Pascal, to program their computers.

## Language Translators

Since the only programs that the computer can run directly are in machine language, programs written in high level languages such as Pascal must be translated into machine language. Programs that do this translation are called language translators. A program that is written in a high level language is called the source code or source program. The machine language that is produced by the translator is called the object code. There are two primary kinds of language translators used in computers, interpreters and compilers.

## Interpreters and Compilers

The major difference between MacPascal and other Pascals is that MacPascal is interpreted, whereas most other Pascals are compiled. Interpreters are computer programs that translate each statement of a source program into object code, one at a time. After each statement is translated it is then executed. Compilers are also computer programs that translate programs written in high level languages into machine language. First the compiler translates the entire source program into machine language, then the program is executed. The advantage of a compiler over an interpreter is that a program translated using a compiler will run several times faster than a program translated using an interpreter. The advantage of an interpreter over a compiler is that an interpreter provides more interaction with the user and makes it easier to develop and correct programs.

## The Operating System

The Macintosh has many resources such as memory, input/output devices, disk, and so on. An operating system is a program that manages these resources and provides access to them for the user and programs. The operating system, for example, contains the programs that allow the user to print a document on the printer or store a document on a disk. The Macintosh has an innovative and easy to use operating system.

## The Macintosh

The Macintosh computer differs from other computers in several respects.

The most obvious characteristic of the Macintosh is its small size, high performance, and relatively low price. Only a generation ago a machine with this kind of performance would take up a whole room and cost several hundred thousand dollars.

Another notable characteristic of the Macintosh is its high resolution video display that is 512 dots wide by 342 dots high. Unlike many other personal computers, the Macintosh uses a bit mapped video display for everything it displays on the screen. A bit is the smallest indivisible unit of memory in a computer. A bit



can be thought of as a switch that can be in one of two discrete states, either 0 (off) or 1 (on). In a bit mapped video display every dot on the screen, called a pixel (picture element), is black or white depending on whether the value of a specific bit in memory is 0 or 1. A bit mapped display allows a program to control exactly what is displayed on the screen down to a single dot. This is a major factor in allowing the Macintosh to mix pictures and text freely on the screen.

The Macintosh mouse is an input device that allows the user to choose easily between options displayed on the screen, or to input spacial information not easily entered using traditional input devices such as the keyboard.

## Macintosh Software

The Macintosh provides a consistent and easy way to use the computer. The computer user no longer must remember many obscure commands in order to make the computer do its job. The Macintosh provides an environment modeled after a desktop, an environment that is familiar to even inexperienced computer users. Like a desktop, documents on the display screen can be moved around graphically by just a simple touch using the mouse and its pointer. The Macintosh software presents computer elements with graphics symbols called icons that are familiar from our everyday lives. The built-in familiarity makes learning how to use Macintosh software a short and easy task. All software written for the Macintosh that follows the Macintosh user interface will be easy to use for people who have learned other Macintosh applications. Macintosh's ease of use provides a major step forward in making computerized problem-solving accessible to the majority of people.

In the following chapters we will examine how the power, speed and versatility of computers can be harnessed using the Pascal programming language.



## 2 Using MacPascal

Before delving into the Pascal language itself let's examine the programming environment by entering and running a MacPascal program. The concepts involved are simple and you will quickly learn how to handle MacPascal like an expert. This chapter will serve as a handy reference later on.

### Getting Up and Running

Turn on your computer, then insert your MacPascal disk into the disk drive slot with the label facing up. After a few seconds the Macintosh desktop will appear on the screen.

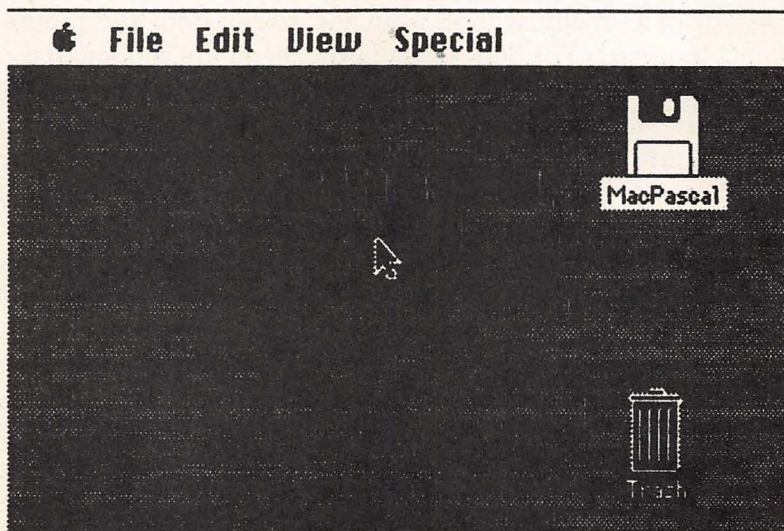


Figure 2-1.



There are several things on the screen to notice.

**The MacPascal Disk Icon.** The picture of a little disk in the right hand corner of the screen. This represents the disk in the disk drive. We will deal more with this in a minute.

**The Pointer.** The arrow you see on the screen is controlled by the mouse. Move the mouse and notice how the pointer moves too. Take a few minutes to get yourself accustomed with the relationship between moving the mouse and the movement of the pointer.

**The Menu Bar.** The list of words across the top of the screen is the names of menus containing commands.

**The Trash Can.** The little picture of a trash can is where things are thrown away.

We are now ready to bring up MacPascal. Move the pointer on top of the MacPascal disk icon and click the mouse button. This is one of the basic operations on the Macintosh called selecting. Notice that when an icon is selected it turns black. Now move the pointer on to the word **File** in the menu bar and hold down the mouse button. You will see a pull down menu appear on the screen, listing several options you can perform on the selected item (Figure 2-2).

The menu will stay on the screen until the mouse button is released. The dark items in the menu are actions that can be performed now. The dim items are actions that would be appropriate under different circumstances. Still holding down the mouse button, pull the pointer down through the menu by moving the mouse. As you move the pointer over the dark items they become inverted (white letters on a black background). Move the pointer over **Open** and select it by releasing the mouse button. You can tell that a menu item has been selected because it will briefly flash before the menu disappears (Figure 2-3).

You have just opened the MacPascal disk. In a few seconds, a window will appear showing icons for the documents and programs contained on the MacPascal disk. To run MacPascal select and open the MacPascal icon with the method described above or use this short cut. After placing the pointer on the icon, click the mouse button twice in very quick succession. Double clicking selects and opens an icon automatically. In about 15 seconds

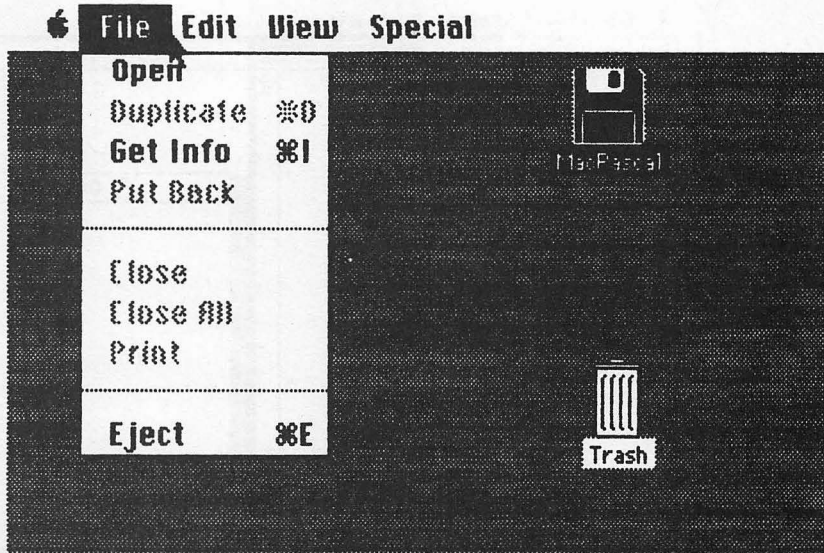


Figure 2-2.

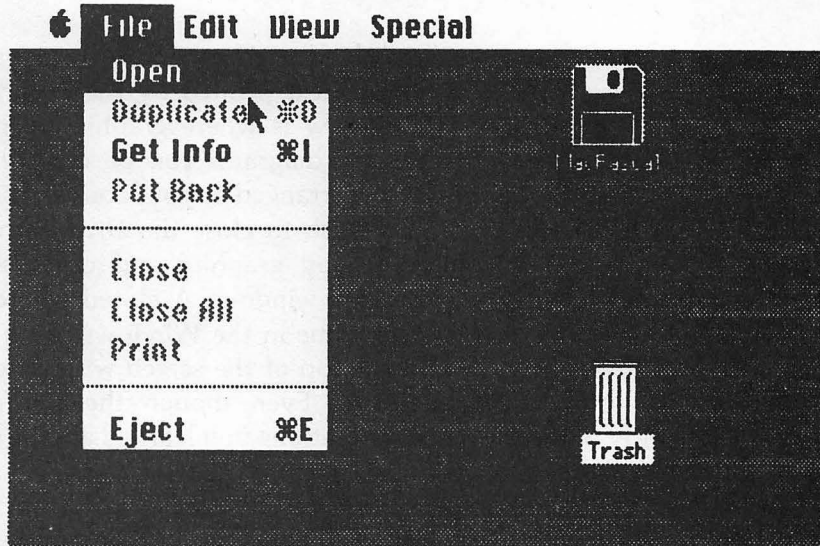


Figure 2-3.

three windows will appear on the screen labeled **Untitled** (the **Program** window), **Text** and **Drawing** (Figure 2-4).

The **Program** window is where you will enter the MacPascal program. It will be titled **Untitled** or titled with the name of a



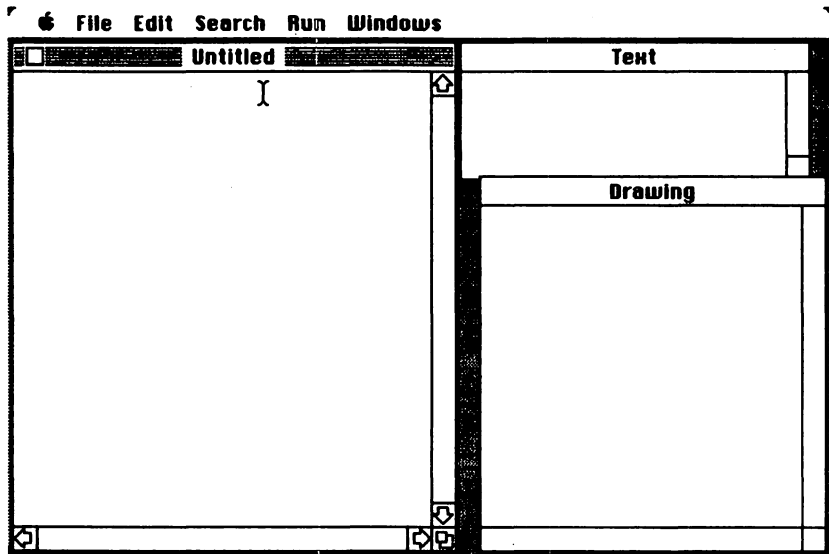


Figure 2-4.

program stored on the disk, depending upon the circumstances. The **Text** window is where text printed by the program is displayed. The **Drawing** window is where graphics such as lines, shapes, graphs, charts, and diagrams can be displayed. These windows can be closed or rearranged at your convenience.

For instance, you may wish to close the **Drawing** window if your program doesn't display graphics and use the space to expand the **Text** or **Program** window. A closed window can be re-opened by selecting its name in the **Windows** menu. You will also see a menu bar on the top of the screen with different titles than those in the desktop. Even though the screen appears slightly different, all the operations you learned are the same as in the desktop.

Locate the pointer on the screen. When you move it into the **Program** window it changes shape. Place the pointer in the upper left hand corner of the **Program** window and click the mouse button. A blinking vertical line called the cursor will appear at that spot. The cursor indicates the current insertion point for new text being typed. Anything typed will be inserted starting at that spot in the window. Any text to the right of that spot will be moved over. Now that the cursor is waiting for text to be entered, let's enter a program. Type the following exactly:

```

program FirstTime;
var
  I : Integer;
begin
  for I:=1 to 10 do
    FrameOval(10,10,5*I,5*I)
  end.

```

If you make a mistake as you type, the backspace key will erase one character to the left. As you enter the program, MacPascal will automatically indent the program and make certain words boldface to conform to MacPascal conventions. Here is how the **Program** window should appear.

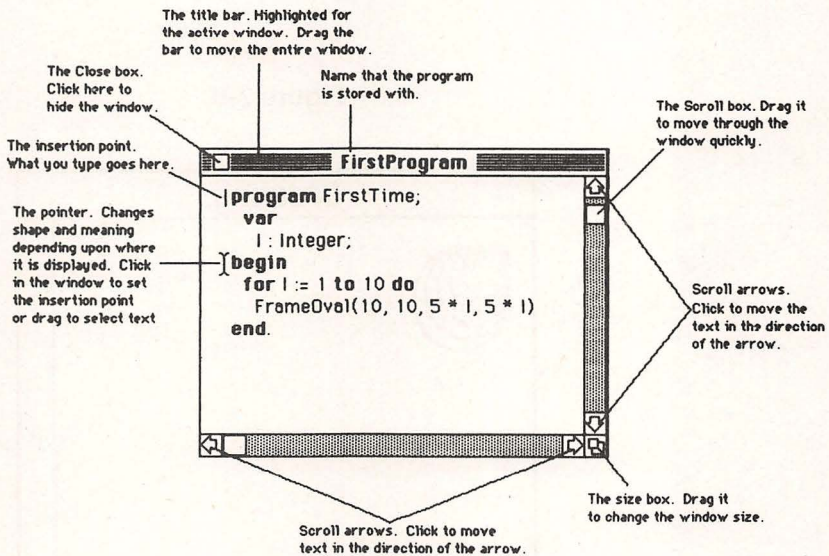


Figure 2-5.

If you made a mistake while typing, turn to the section of this chapter concerning editing a program.

Once you are satisfied that you typed the program correctly run the program by selecting **Go** from the **Run** menu (Figure 2-6).

The program will display tangential circles in the **Drawing** window as is shown in Figure 2-7.

How this program works is not overly important at this point. You will quickly learn how it is done. What is important is that



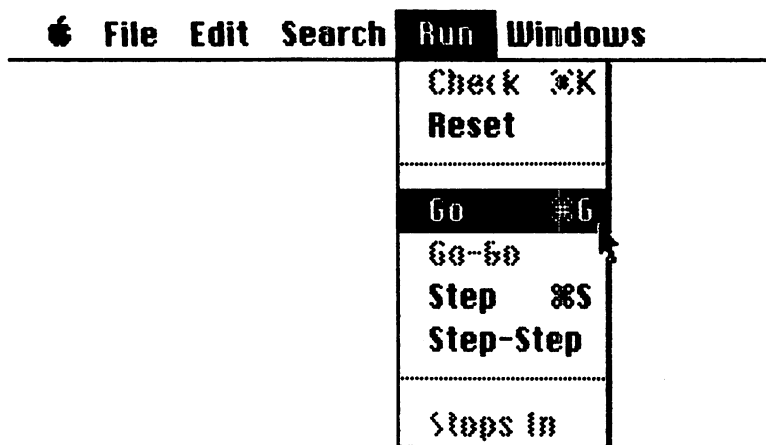


Figure 2-6.

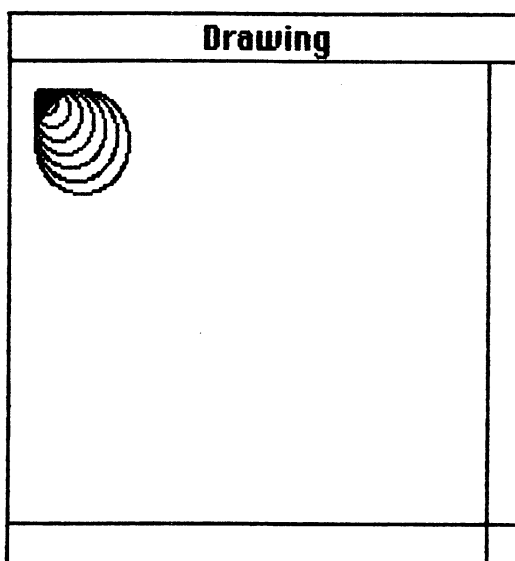


Figure 2-7.

you entered the program correctly and that the program worked as expected. If the program did not work as expected, check the program carefully and make sure that it is exactly the same as the example.

## Editing a Program

If a mistake is made while entering a program or a change in the program needs to be made, the following editing techniques are available.

### *To Insert Text*

1. Move the pointer to the desired spot in the program and click the mouse button to place the insertion point.
2. Enter the text.

### *To Delete Text*

There are two different techniques for deleting text.  
For a small amount of text.

1. Move the pointer to the right of the characters you want to delete and click the mouse button to place the insertion point.
2. Use the Backspace key to delete the characters.

For a large amount of text.

1. Select the text to be deleted by placing the pointer at the start of the text to be deleted and hold down the mouse button.



I.|TEXT BEING SELECTED

Figure 2-8.

2. Now drag the pointer across the text. Notice that, while dragging, the area selected is displayed inverted (white characters on a black background). At the end of the portion to be deleted release the button. The text to be deleted should now be all in reverse.



TEXT BEING SELECTED I

Figure 2-9.

3. Delete the selected area by pressing the Backspace key.

### *To Replace Text*

1. Select the text to be replaced by using the dragging technique described above.



2. Start typing the new text. The old text is automatically replaced.

### *To Move Text*

1. Select the text to be replaced.
2. Choose **Cut** from the **Edit** menu.
3. Place the insertion point where you want the text to go.
4. Choose **Paste** from the **Edit** menu.

### *To Copy Text*

1. Select the text to be replaced.
2. Choose **Copy** from the **Edit** menu.
3. Place the insertion point where you want the text to go.
4. Choose **Paste** from the **Edit** menu.

### *Shortcuts*

1. Double clicking the mouse button on a word automatically selects the word.
2. Triple clicking the mouse button on a line automatically selects the entire line.

These editing procedures are exactly the same as those used in MacWrite and in many other places in the Macintosh such as the Notepad.

## **Saving a Program**

Once you are finished using a program you can save it on a disk for future use. To save the program you are currently working on select **Save As . . .** from the **File** menu. A dialog box will appear on the screen (Figure 2-10).

You must give your program a name before you save it on the disk. This will be the name used to store the program and to retrieve it later on. This name is unrelated to the identifier in the **program** statement in the program. Enter a name for the program by typing it into the long box where the cursor is blinking. Give the program a name that will represent what it does, but don't hit the Return key yet.

Notice the three ovals in the dialog box containing **Save**, **Cancel**, and **Eject**. These are known as buttons and they are pressed by clicking the pointer inside the oval. Clicking on **Save**



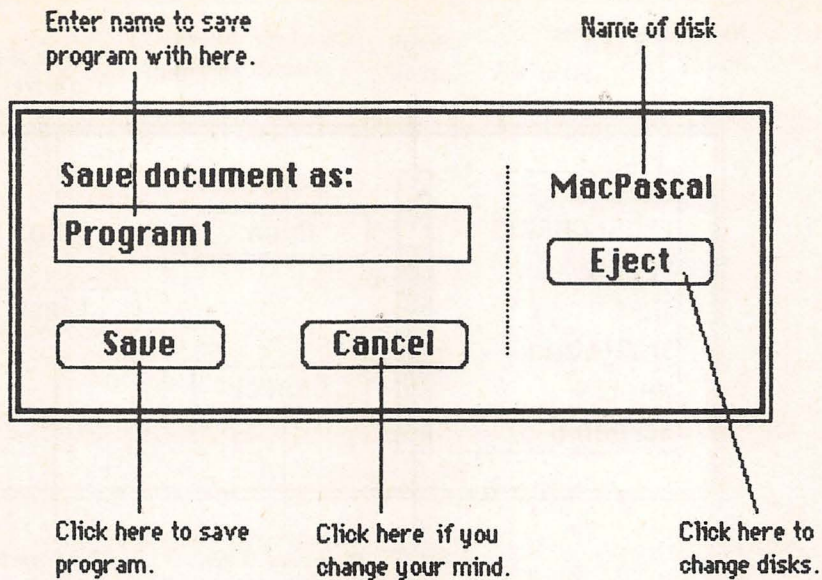


Figure 2-10.

or pressing the Return key will save the program on the disk in the disk drive. Clicking on Eject will eject the disk currently in the drive allowing you to insert a different disk to save your program on. After the program is saved, MacPascal will prompt you to re-insert the original disk. The **Cancel** button can be used if you change your mind about saving the program.

If a program already has a name from being saved at an earlier time it can be quickly saved again by selecting **Save** from the **File** menu rather than **Save As . . .**. If the program was originally saved on a different disk MacPascal will eject the disk in the disk drive and prompt you to enter the other disk.

## Recalling a Program

You can recall a program that is saved on a disk in either of two ways.

If MacPascal is already running then select **Open . . .** from the **File** menu. A dialog box will appear on the screen (Figure 2-11).

The names of the MacPascal programs that are stored on the disk appear in a small window on the left side of the dialog box. If more programs exist than can be listed in the space, the same scrolling apparatus as in the **Program** window will also appear.



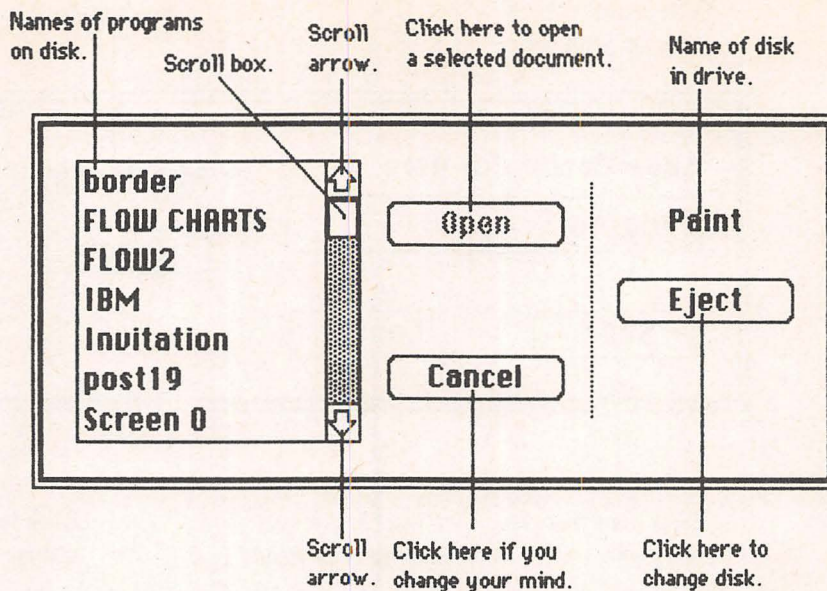


Figure 2-11.

To recall a program, either select the program name by clicking on it and then clicking on the **Open** button or just double click on the program name. A program on a different disk can be opened by ejecting the disk currently in the disk drive and inserting the disk containing the program. The programs contained on the new disk will then appear in the window. Alternatively, when you are in the desktop you can recall a program by opening the icon for that program. This will automatically bring up MacPascal and then load the program.

## Printing a Program

A copy of your MacPascal program (the text itself, not the output) can be printed on the printer connected to your Macintosh. Select **Print** from the File menu. A dialog box will then appear on the screen (Figure 2-12).

Several buttons allow different printing options to be specified. High quality printing is slow but produces a copy of the program in impressive print quality. Standard prints faster but the print quality is lower. Draft is the fastest printing mode but the output may not be an exact copy of the screen (the text is printed in a different font). The boxes for the number of copies and the range

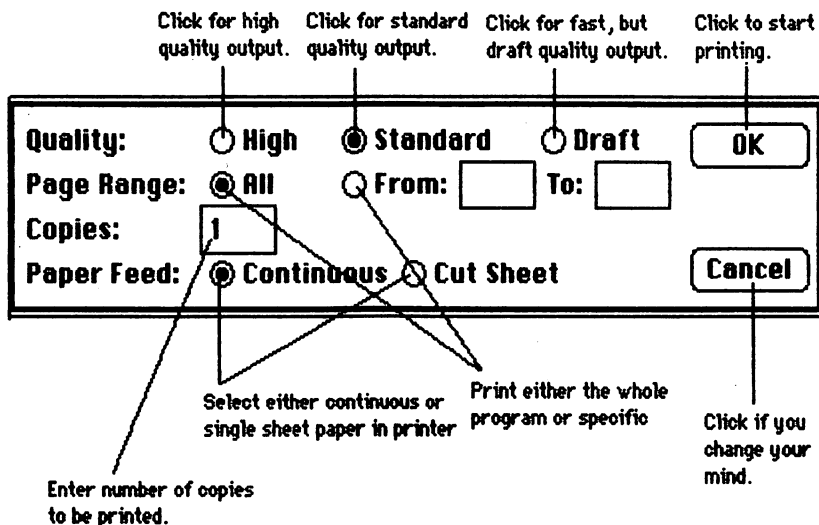


Figure 2-12.

of pages to print can be filled in by placing the cursor in the box. The Tab key will move the cursor from box to box without the use of the mouse.

## Printing the Active Window

The contents of the active window can be printed by simultaneously holding down the Shift, Command, and the number "4" keys. You can make any window the active window by clicking the pointer anywhere inside of it.



# 3 Pascal Fundamentals

The best way to go about learning Pascal is to dive right in, so here is our first program.

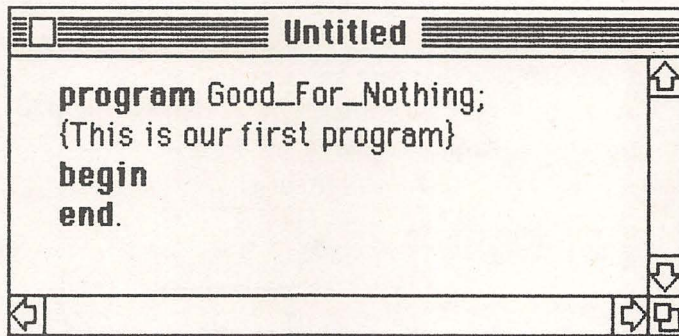


Figure 3-1.

This is the simplest program possible in Pascal. It does absolutely nothing. All programs must begin with the word **program** to identify them as programs. Every program must also have a name separated from the word **program** by a space. This program's name is "Good\_\_For\_\_Nothing". A name in Pascal is called an identifier. In MacPascal, identifiers can be made up of letters, numbers, and underscores. An identifier can contain up to 255 characters. In practice, an identifier will contain from one to about fifteen characters (who wants to type very long names?). It is good practice to use identifiers that have a meaning that corresponds to the functions they perform or the objects they represent.

## Syntax

Syntax is the rules for constructing valid statements in a language. In natural languages such as English, we call these rules

grammar. Programming languages also have rules of syntax, but they are much more restrictive and simpler than natural languages. Pascal's syntax is so simple and well defined it can be expressed in a series of diagrams. For instance, here is the syntax diagram for a digit.

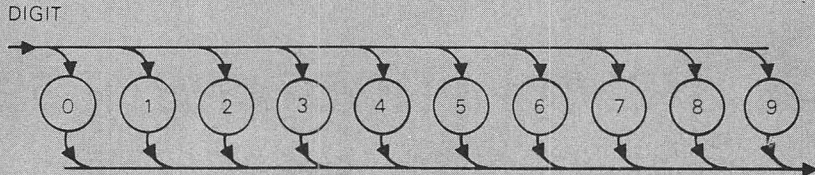


Figure 3-2. Syntax Diagram : Digit.

To construct a syntactically correct digit, follow the diagram from the starting point on the left to the end point on the right. Many paths are possible. Any path that starts on the left and ends at the right describes a digit that is syntactically correct. Notice in this diagram any single digit from 0 to 9 is a valid digit. A slightly more complex syntax diagram is that of an unsigned integer (a whole number).

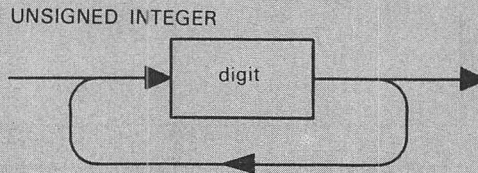


Figure 3-3. Syntax Diagram : Unsigned Integer.

Once again any path that goes from the start to the end defines a syntactically correct unsigned integer. To confirm that a number such as 327 is a valid unsigned integer, we would follow the diagram traversing the middle section three times and then exit.

The syntax diagram for a signed integer is built with the rules for an unsigned integer:

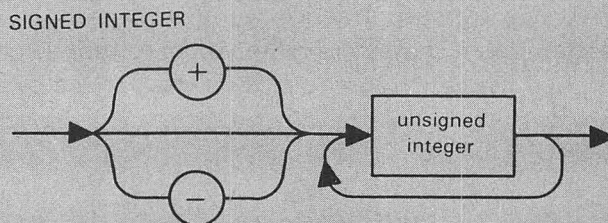


Figure 3-4. Syntax Diagram : Signed Integer.



In a diagram, the elliptical symbols are atomic, they can't be sub-divided into smaller diagrams. Rectangular symbols are those that can be sub-divided. For example, in the signed integer diagram, the unsigned integer symbol was defined previously. A complete listing of the syntax diagrams for Pascal can be found in Appendix D.

Not all words can be used as identifiers. A numeric digit or an underscore cannot be the first character of an identifier. Spaces, punctuation characters, and other special characters cannot be used in identifiers either. The following are valid identifiers:

Big\_\_Bucks    Total    Rate3    One4all

The following are not valid Pascal identifiers:

22go    Big:Bang    Counter\$

Lower case and upper case characters are equivalent in an identifier; therefore, the following two identifiers are treated as one and the same by MacPascal.

GrossPay    grosspay

The underscore character is not in the identifier and is often used as a separator between words to make identifiers easier to read. The following identifiers are considered to be different by MacPascal:

Gross\_\_Pay    GrossPay

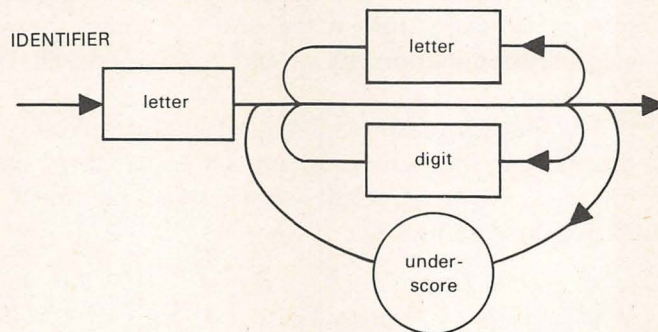


Figure 3-5. Syntax Diagram : Identifier

The word **program** cannot be used as an identifier because it has a special meaning in the language. Words like **program** that



have special meanings are called "reserved words." or "word symbols". There are 70 reserved words in the MacPascal language which are listed in Appendix F. In MacPascal, and in this book, reserved word are displayed in boldface type.

Following the reserved word **program** and the program name identifier is a semicolon. Semicolons are used in Pascal to separate statements. A statement is a complete Pascal instruction.

## Comments

Information that is enclosed between curly braces { and } are called comments. Comments are used to document the workings of a program and are meant for people to read. The Pascal interpreter completely ignores all comments. It cannot be emphasized too strongly that comments are required for writing clear maintainable programs that can be understood when examined later. A parenthesis asterisk pair, '(' and ')' can be substituted for the curly brace pair, '{' and '}'. The start and end of a comment must use matching delimiters—a '{' and '}' or a '(' and ')' but not '{' and '\*' or '(' and '}'.

## Documenting a Program

It is important that when you write a program, you include in the program explanations of how the program works. This may not seem important or necessary at the time, but you will be grateful in the future when you look back at your work. The eleventh commandment should read "Thou shalt document thy programs."

There are two aspects to properly documenting a program. The first is to make your program self-explanatory by using a meaningful identifier name. Below are two assignment statements that perform the same task.

```
X := Y + Z;  
SalePrice := Price + Tax;
```

The only thing that can be ascertained from the first statement is that two variables are being added together and the result assigned to a third. From the second, the reader can tell the reason why the statement is being executed.



The second aspect of documentation is the use of comments. These should be used when the identifier names alone can not indicate the purpose of the statement or group of statements or to describe what a group of statements is doing.

```
const
ConversionRate = 21374; {# of Lire in a Dollar}
:
:
{Calculate import cost}
Dollars := Lire * ConversionRate;
Duty := Dollars * TaxRate;
Cost := Dollars + Duty;
```

Appendix C contains an entire program meticulously documented to serve as a guide.

After the comments in Figure 3-1 are the reserved words **begin** and **end**. All programs contain a **begin** and finish with an **end**. The **end** is followed by a period indicating that it is the final statement of the program.

## Write and Writeln

Now let us look at a program that does something. For a computer to be useful it must be able to output information for people to see. The two most common statements in Pascal that output information are the **Write** and **Writeln** (pronounced write line) statements.

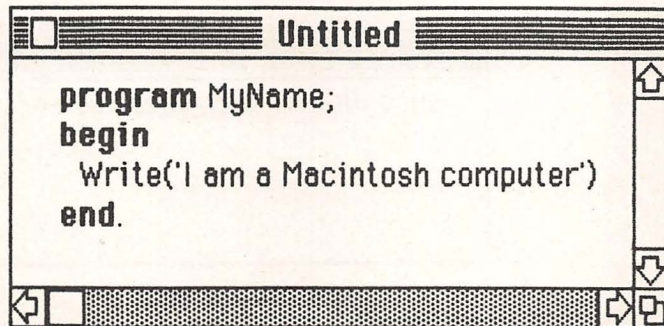


Figure 3-6.



Program MyName will display in the Text window the sentence

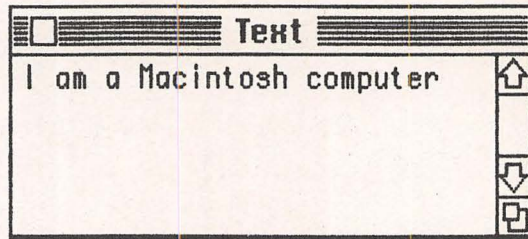


Figure 3-7.

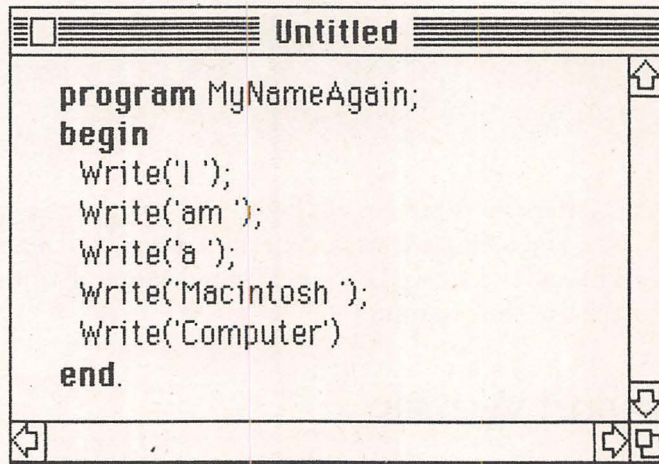


Figure 3-8.

Program MyNameAgain does the same exact thing. Successive Write statements will place information on the same line in the Text window.

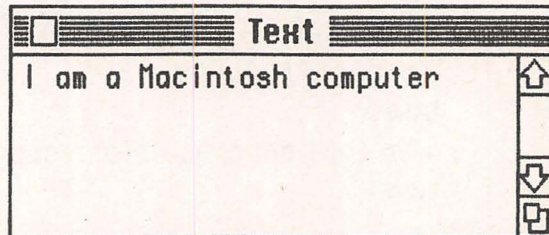


Figure 3-9.

The following program will yield slightly different results:



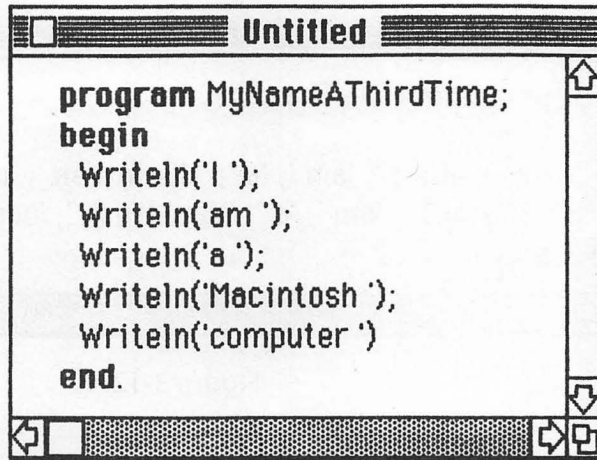


Figure 3-10.

Program MyNameAThirdTime displays

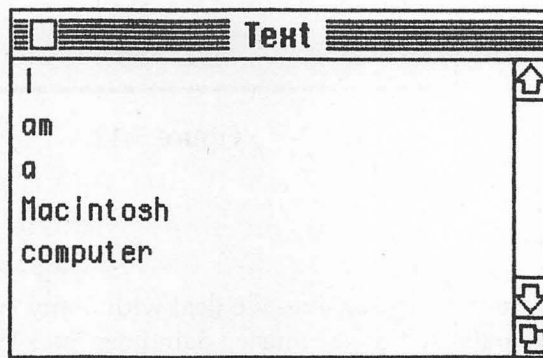


Figure 3-11.

This program would display each word on a separate line. The difference between `Write` and `Writeln` is that whatever is displayed after a `Writeln` statement is displayed on the line below. Whereas, whatever is displayed after a `Write` statement is displayed on the same line. Program `MoreThanOne` produces the output shown in Figure 3-13.

More than one item can be used in a `Write` or `Writeln` statement as long as each item is separated by commas. `Write` and `Writeln` can be mixed in the same program.

Now that we have a way to display information let us examine the kinds of information Pascal can handle.

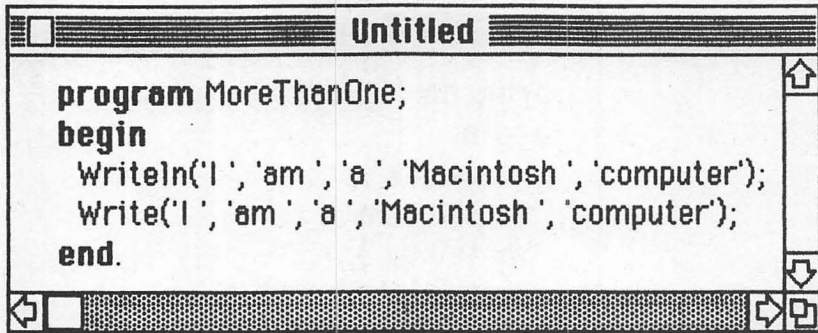


Figure 3-12.

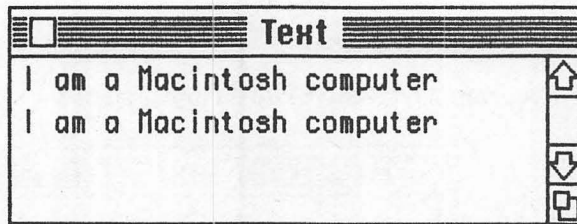


Figure 3-13.

## Data

In our everyday lives we deal with many types of information: newspapers, books, music, paintings, etc. When information is used in the computer it is called data. Computers process many different kinds of information or data, including numbers, characters, strings of characters, pictures, sounds, etc.

## Data Types

There are several different types of data that can be represented in Pascal: Integers, Char, String, Boolean, Real, and so forth. Integers are positive and negative whole numbers (numbers that don't have fractional parts). Examples of integers are:

1 23 -252 0 1398 -12

Notice that integers contain no decimal points. Commas are not used or allowed in integers. The largest integer that can be



used by the Pascal language is called Maxint. This number is dependent on the machine on which Pascal is being used. On the Macintosh system Maxint is 32767. The smallest number that can be represented is negative Maxint or  $-32767$ . It is also desirable to be able to represent numbers that have a whole part and a fractional part. Pascal lets us represent these numbers with the Real data type. Examples of real numbers are:

3.14   -87.0   242.34   1.324e+6   -7.43e-2

The first three numbers listed are in the notation with which you are familiar, that is, numbers with digits to the right of the decimal point. The last two numbers are in a notation called floating point notation. The "e" stands for exponent. The number is interpreted as the number on the left of the "e" multiplied by ten raised to the number following the "e". The exponent always has a sign. This is sometimes also called scientific notation. The number  $1.324e+6$  is equivalent to  $1.324 \times 10^6$  or 1324000.0. The number  $-7.43e-2$  is equivalent to  $-7.43 \times 10^{-2}$  or  $-0.0743$ . Other examples are:

|            |                  |            |
|------------|------------------|------------|
| -12.34e+2  | is equivalent to | -1234.0    |
| 34.567e+1  | is equivalent to | 345.67     |
| -932.13e-4 | is equivalent to | -0.0932113 |

Real numbers can be written in standard fashion or in floating point notation. There must be at least one digit preceding the decimal point in a real number. 0.5 is a valid representation of one-half whereas .5 is not. Real numbers are useful for representing very large or very small quantities since their range of possible values is far greater than that of integers.

Computers need to process information other than numbers in order to communicate effectively with people. Pascal has two data types that allow us to use text information with our computers, the character and the string data types.

Characters are upper- and lowercase letters, numbers and punctuation symbols. Character data is enclosed in single quotes. Examples of characters are the following:

'a' 'D' '!' ';' '3' '%' ' '

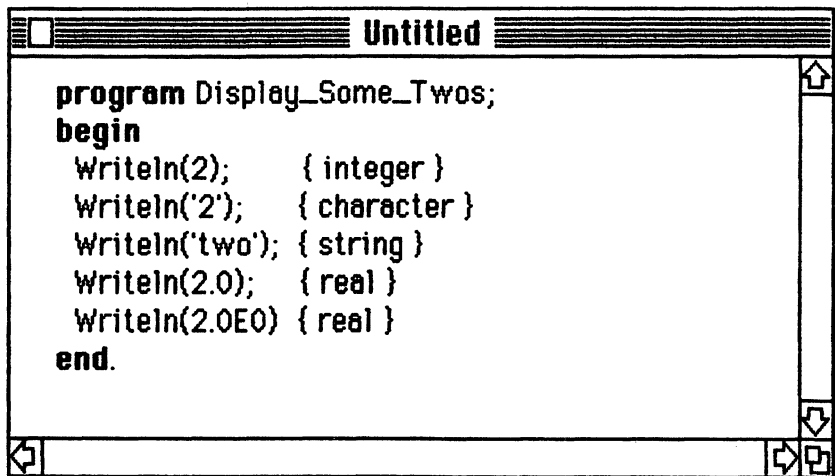
Since quotes are used to delineate a character, we must use a different method to represent the quote character itself. The quote character is represented by two consecutive quotes between quotes for a total of four separate quotes together:

""

Strings are sequences of characters and are useful because they let us combine individual characters into words or sentences. This lets us manipulate more meaningful units, rather than treating characters individually. As with characters, strings are written between single quotation marks:

```
'This is a sample string'  
'testing 123 testing!!!'
```

Lets look at a simple Pascal program that uses the above data types:

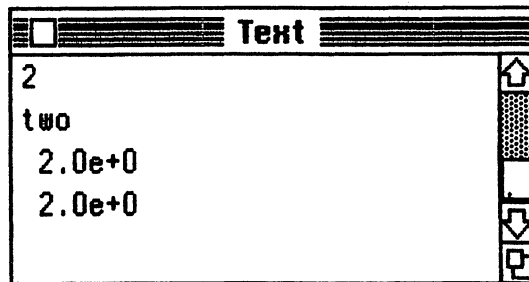
A screenshot of a Pascal program editor window titled "Untitled". The window contains the following Pascal code:

```
program Display_Some_Twos;  
begin  
  Writeln(2);    { integer }  
  Writeln('2');  { character }  
  Writeln('two'); { string }  
  Writeln(2.0);  { real }  
  Writeln(2.0E0) { real }  
end.
```

The code is displayed in a monospaced font. The window has a standard Mac OS-style title bar and scrollbars.

Figure 3-14.

The output of Display\_Some\_Twos is

A screenshot of a text window titled "Text". The window displays the output of the program from Figure 3-14, with each result on a new line:

```
2  
two  
2.0e+0  
2.0e+0
```

The window has a standard Mac OS-style title bar and scrollbars.

Figure 3-15.

The Writeln statement prints to the Text window whatever data is between the parentheses. After a Writeln statement is exe-



cuted whatever is printed next will be printed on the following line of the Text window. Each piece of data is printed on a separate line. Line three of the program prints out the integer 2. Line four of the program prints out the character 2. Notice that the quotation marks around the character 2 are not printed out and are used to indicate that this 2 is a character. Line 5 prints out the string "two". Line six and seven show that the two different floating point representations of 2.0 are equivalent.

## Variables

Each piece of data used is stored in the computer's memory. The computer's memory can be thought of as being divided into many different compartments, each holding a piece of information. There is a unique numeric address corresponding to each compartment. The address allows programs to identify and access the data stored in the compartment. Fortunately, in Pascal the numeric address of each compartment is transparent to us. Instead of using numeric addresses, Pascal has words called identifiers that represent each compartment. Just as people have names, each piece of information in a Pascal program can have a name.

Variables can be thought of as these compartments in the computer's memory where data is stored. One important feature of variables is that the value of the data kept in the compartment can change during the execution of a program. Pascal permits us to have variables for all the data types defined above. In Pascal programs we reference a variable and the data contained within it by using an identifier that is associated with the variable (Figure 3-16).

In the above program two variables were declared: Number, which contains an integer, and Ch, which contains a character. Variables are declared after the program statement but before the first **begin** of the program. The start of the variable declaration section is indicated by the reserved word **Var**. A variable declaration statement consists of the variable's identifier and the type of the variable, separated by a colon (:). A semicolon separates each of the variable declaration statements.

Program TestTwo simply writes on the screen the integer value contained in the variable 'Number' followed by a string of 3 spaces (as indicated by the three spaces in quotes) and then the



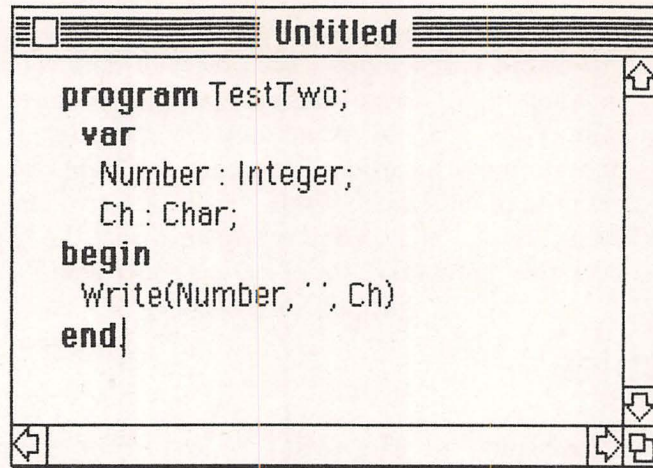


Figure 3-16.

characters contained in the character variable 'Ch'. The output of this program is as follows:

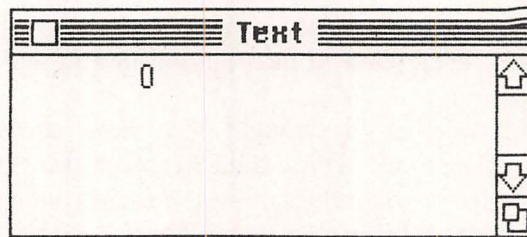


Figure 3-17.

Why did the computer display the integer zero followed by nothing? The reason is that these are the values that MacPascal places in variables when a program is first run. An integer gets an initial value of 0, while a character is initialized with a blank (hence nothing was displayed for the variable Ch).

In other implementations of Pascal the results may be less predictable. Other implementations would display whatever happened to be in memory from the last program that was run on the machine. When writing programs it is good practice not to rely on the computer to give values to variables. Initializing a variable means giving it a value before it is used. The following program initializes the variables then displays them.



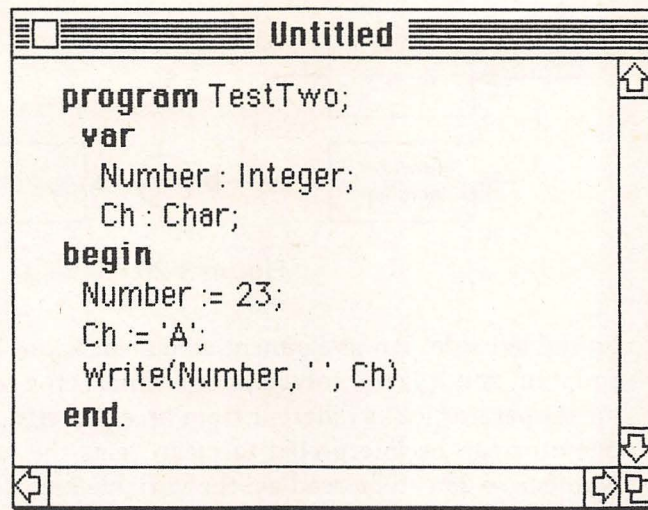


Figure 3-18.

After this program is run the following will be displayed in the Text window:

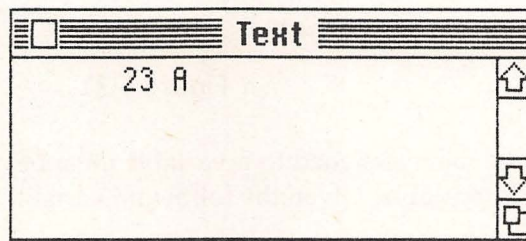


Figure 3-19.

## Assignment Statements

A statement like "Number := 23" is known as an assignment statement. The syntax of an assignment statement is indicated in Figure 3-20.

The value on the right hand side of the := is assigned to the variable on the left hand side. This is used to place a value into a variable. The variable retains the assigned value until the variable is altered by some other statement in the program. The ":=" is known as the assignment operator. Notice that there is no space between the colon and the equal sign. Only one variable can be

## ASSIGNMENT STATEMENT

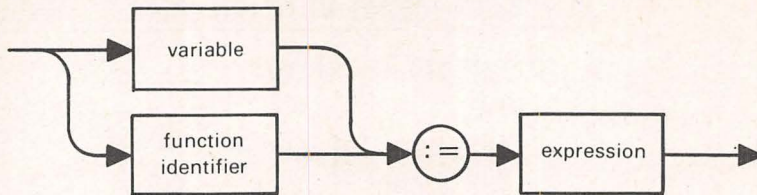


Figure 3-20.

on the left side. An assignment statement is not like an algebraic equation and it is not solved as one. This is the reason the assignment operator looks different from an equal sign. The assignment operator can be interpreted to mean "gets the value of". Hence, `Number := 23` can be read as: the variable `Number` gets the value 23. It may be helpful to think of the integer value 23 being placed into the memory compartment which is labeled `Number`.



Figure 3-21.

The value assigned to a variable must be of the same data type as the variable. Given the following variable declarations:

```

var
  I : Integer;
  R1, R2 : Real;
  Ch : Char;
  
```

The following assignment statements are all legal.

```

I := 17;
R1 := 2.03;
R2 := 15.0;
Ch := 'B';
  
```

The following are illegal assignment statements.

```

I := -17.17;    -17.17 is not an integer
Ch := 22;       Ch is a character variable
  
```



## More on Write and Writeln

The Write and Writeln statements have an optional parameter called the fieldwidth parameter to provide control over how data is displayed in the Text window. The fieldwidth parameter is specified by following any item in a Write or Writeln statement with a colon (:) and a positive integer. The integer determines how many spaces are allocated for display of the data item. The easiest case to look at is the display of strings. The fieldwidth parameter indicates how many spaces are used to display the string. If the string does not take up all the spaces allocated, it is right justified within the field as demonstrated in Figures 3-22 and 3-23.

Write('Cantaloupe' : 15)

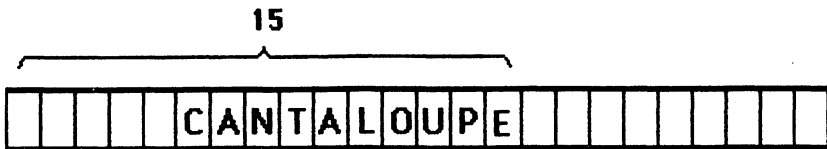


Figure 3-22.

Write('Cantaloupe':20)

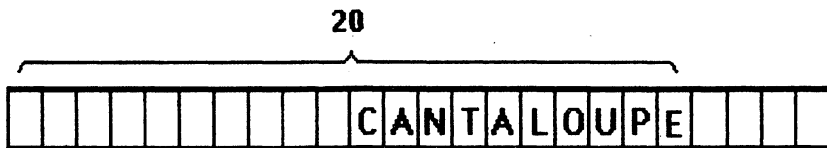


Figure 3-23.

If the number of characters in the string exceeds the fieldwidth parameter the excess characters on the right side are truncated. Figure 3-24 demonstrates this:

Write('Cantaloupe':5)

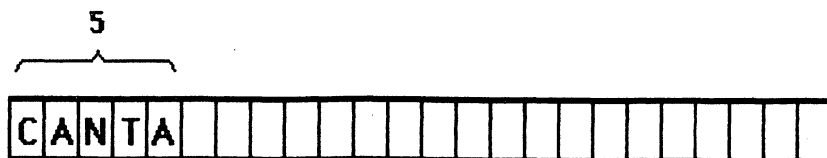


Figure 3-24.

If a fieldwidth parameter is omitted the string parameter is printed in precisely the number of characters required. When displaying integers the fieldwidth parameter is used in a similar fashion as with strings but with two differences. If the fieldwidth parameter is omitted, the integer is displayed right justified in a minimum of eight spaces. If the integer is too long to fit in eight spaces, the exact space needed is allocated. An integer value will never be truncated.

In order to display real numbers in standard notation two fieldwidth parameters are used, the first for the total field width not counting the decimal point and the second for the number of digits after the decimal point. Consider the following assignment statement:

```
R := 2.55;
```

the following `Writeln` statement

```
Writeln(R:4:2)
```

displays the following:

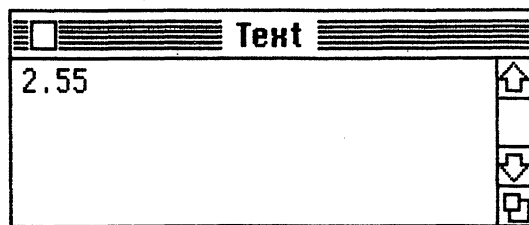


Figure 3-25.

If the total field width given is too small for a Real number or an Integer the entire value will be printed anyway.

```
I := 10;  
R := 10.04;
```

The following `Writeln` statements:

```
Writeln(I :1);  
Writeln(R : 3 : 2);
```

will display



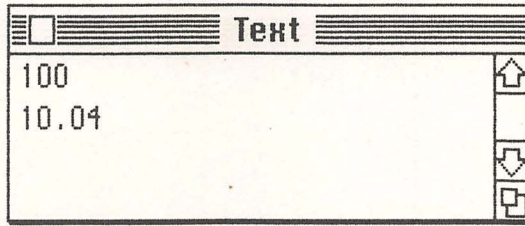


Figure 3-26.

In writing a Real, the value will be rounded off to the number of decimal places specified by the second field width parameter.

Consider the following assignment statement:

```
R := 9.046
```

the following `Writeln` statement

```
Writeln(R : 5 : 2);
```

will display

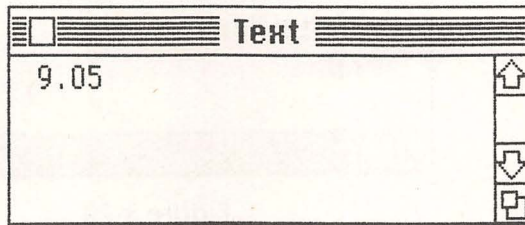


Figure 3-27.

The value was rounded off to two decimal places.

When no fieldwidth parameter or one fieldwidth parameter is used with real numbers they are displayed in floating point notation.

Consider the following assignment statement:

```
R := 12.03;
```

The following statement

```
Writeln(R);
```

will display Figure 3-28.

## Expressions

Besides assigning constant values to variables we can also assign the values of arithmetic expressions to variables (Figure 3-29).

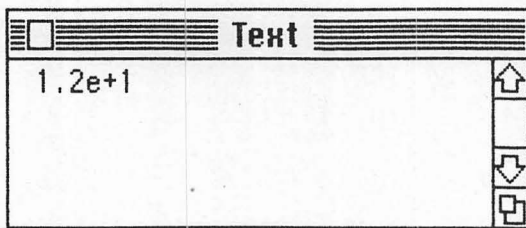


Figure 3-28.

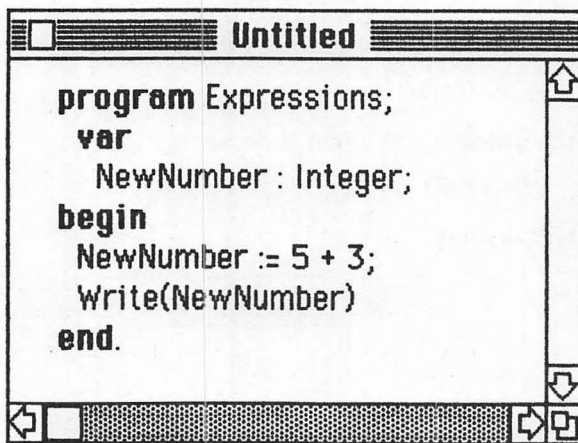


Figure 3-29.

In this program the variable "NewNumber" is assigned the value that is obtained by first evaluating the expression on the right hand side of the assignment operator ( $:=$ ). The assignment statement can be read as: the variable NewNumber gets the value obtained by adding  $5 + 3$ . The value that is printed out, of course, is 8.

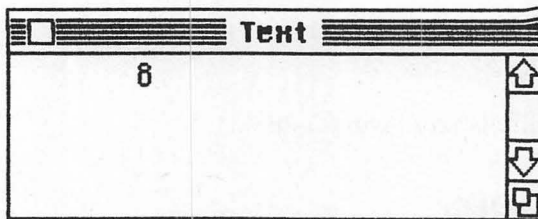


Figure 3-30.

An expression can contain variables as well as constants.



```
NewNumber := OldNumber + 17;
```

This statement adds 17 to the value of the variable OldNumber and assigns that value to the variable NewNumber. There is no change in the value of OldNumber since it appears on the right side of the assignment operator. Only a variable on the left side of the assignment operator is changed.

The value of one variable can be given to a second variable in the same way.

```
NewNumber := OldNumber;
```

Here the value of OldNumber is assigned to NewNumber. They will both now have the same value. An assignment statement that might appear unclear at first sight is where the same variable appears on both sides of the assignment operator.

```
Number := Number + 3;
```

What is important to remember is that assignment is very different from an algebraic expression. This statement simply adds 3 to the value contained in the variable Number. This statement is analyzed just like any assignment statement. The expression on the right side is evaluated and assigned to the variable on the left side. That is, add 3 to the value of Number and place that back into Number. Another way of describing this statement is: the "new" value of Number is the "old" value of Number plus 3.

Addition is not the only operator operation that can be used in an expression. There are several other arithmetic operators available in Pascal:

|            |   |
|------------|---|
| +          | real or integer addition                        |
| -          | real or integer subtraction                     |
| *          | real or integer multiplication                  |
| /          | real division                                   |
| <b>div</b> | integer division                                |
| <b>mod</b> | modulo division (remainder of integer division) |

The \*, +, and - operators work as expected on both integer and real numbers but different divisions exist for real and integer values. The div operator is used to divide one integer by another. When we divide integers the remainder is discarded. Some examples:

| <u>Expression</u> | <u>Value</u> |
|-------------------|--------------|
|-------------------|--------------|

|         |   |
|---------|---|
| 8 div 2 | 4 |
| 8 div 3 | 2 |
| 8 div 9 | 0 |

The **mod** operator is used to find the remainder of an integer division. **mod** does not do the division, it just calculates what the remainder would be. For instance, 10 **mod** 3 is 1, because 10 divided by 3 is 3 with a remainder of 1. Some more examples:

| <u>Expression</u> | <u>Value</u> |
|-------------------|--------------|
|-------------------|--------------|

|         |   |
|---------|---|
| 8 mod 8 | 0 |
| 8 mod 2 | 0 |
| 8 mod 9 | 8 |
| 8 mod 3 | 2 |

Both 8 **div** 0 and 8 **mod** 0 are both illegal since you can not divide by zero.

Real division (the / operator) divides to real values. 5.0/2.0 yields 2.5 and 10.0/1.0 yields 10.0. With any of the operations real values can be mixed with integer values in an expression. When this is done the integer value is automatically converted to a real prior to the operation. For instance, in the addition of 4 + 3.7 an integer and real are both used. The integer 4 will first be converted to the real 4.0 and then added to 3.7 The result is 7.7.

The result of an expression can only be assigned to variables of the same data type. Breaking this rule will cause an error. Given these variable declarations:

```
var
  I, J : Integer;
  X : Real;
```

The following statements are all legal:

```
I := J + 3;
X := I + 2.0;
X := 5 / 2;
X := J;
```

The following statements are all illegal:



$J := Y;$  Can't assign a real value to an integer variable  
 $J := 3.0 \text{ div } 2;$  div needs two integer operands  
 $J := J / I;$  Real division produces a real result

The following chart summarizes the data type of the result of different operators.

| Type of Operand |              |                 |                 |                    |
|-----------------|--------------|-----------------|-----------------|--------------------|
| Operator        | Real<br>Real | Real<br>Integer | Integer<br>Real | Integer<br>Integer |
| +               | Real         | Real            | Real            | Integer            |
| -               | Real         | Real            | Real            | Integer            |
| *               | Real         | Real            | Real            | Integer            |
| /               | Real         | Real            | Real            | Real               |
| div             | error        | error           | error           | Integer            |
| mod             | error        | error           | error           | Integer            |

## Operator Precedence

How is the value of an expression with more than one operator calculated? What is the result of the following operation?

$$7 + 2 * 4$$

If the addition is done first the result is 36. If the multiplication is done first the result is 15. To avoid this type of ambiguous situation, some operations in Pascal have a higher order of precedence than others. Operators with high precedence get evaluated before operators with low precedence. Multiplication, division, div and mod have higher precedence than addition and subtraction; therefore  $2 * 4$  gets first evaluated to 8 and then 7 gets added to yield 15.

### Operator Precedence Table

|                 |                                |
|-----------------|--------------------------------|
| High Precedence | $*, /, \text{mod}, \text{div}$ |
| Low Precedence  | $+, -$                         |

When operators of the same precedence are found in an expression, they are evaluated from left to right.

The natural precedence of the operators can be overcome by the use of parentheses. If we wished to add the 7 and 2 together and then multiply the result by 4 we could express it this way:

$$(7 + 2) * 4$$

Some more examples:

| Expression        | Value |
|-------------------|-------|
| $3 + 2 * 3$       | 9     |
| $(3 + 2) * 5$     | 25    |
| $14 \bmod 3 + 1$  | 3     |
| $1 + 2 * 3 + 4$   | 11    |
| $(1 + 2) * 3 + 4$ | 13    |
| $1 + 2 * (3 + 4)$ | 15    |

## Constants

So far we have assigned constant values to variables. Constants, like the name implies, are values that remain constant. Pascal can also have constants that are represented by identifiers. The following program demonstrates constants:

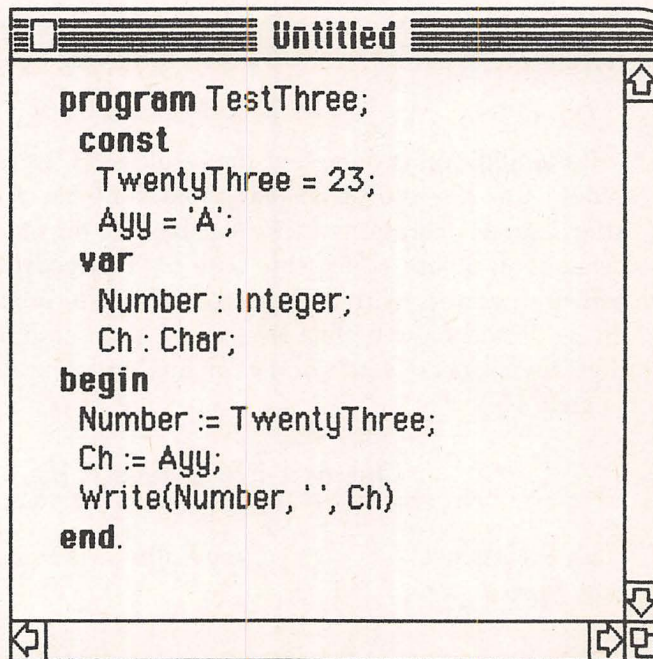


Figure 3-31.



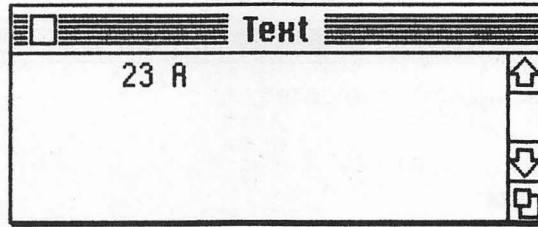


Figure 3-32.

Constants are declared after the program declaration statement but before the variable declaration section. The key word **const** is used to indicate the start of the constant declaration section. A constant declaration statement consists of the constant's identifier and the value of the constant separated by an equal sign (=). A semicolon separates each of the constant declaration statements.

Notice the difference between constants and variables. The identifier associated with a variable is the name of a location in the computer's memory which contains a value. The value contained in that location can change by assigning that location a new value with the `:=` operator. Once a constant is defined it cannot be redefined later in the program. The identifier associated with a constant becomes another name for that constant. An equal sign is used to show that the constant identifier and the value itself are equal and have the same effect. The reason why constant identifiers are used rather than the constant values themselves is that the constant's name can make the program more understandable.

To demonstrate constants, let's write a program that calculates the area of a circle. This requires the value of Pi (3.1415) which will never change and is best represented by a constant (Figure 3-33).

This program is much easier to understand because of the use of the constant Pi but is not very useful unless we want to know the area of a circle whose radius is 5. A more general and useful program would allow the calculation of the area of any circle. Another reason for using constants is their ability to detect errors. Since we know that a constant can never be changed, any erroneous attempt to alter its value will result in a program error indicated when you attempt to run the program. It is far better to detect an error in this fashion than to hunt for the source of improper calculation (Figure 3-34).

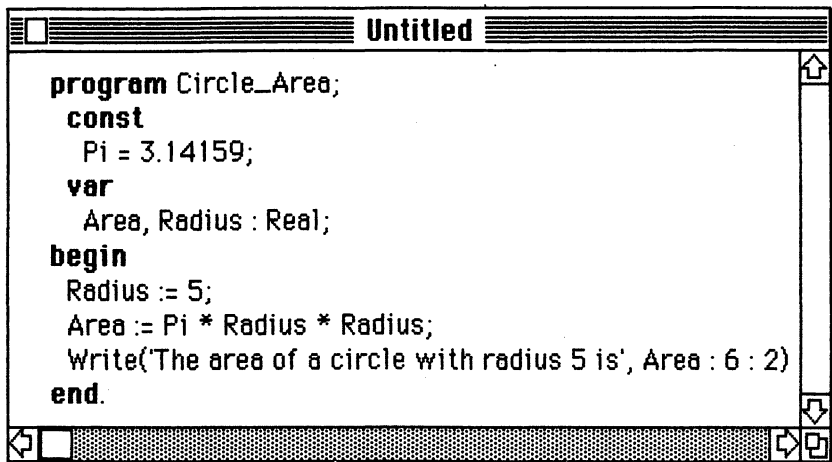


Figure 3-33.

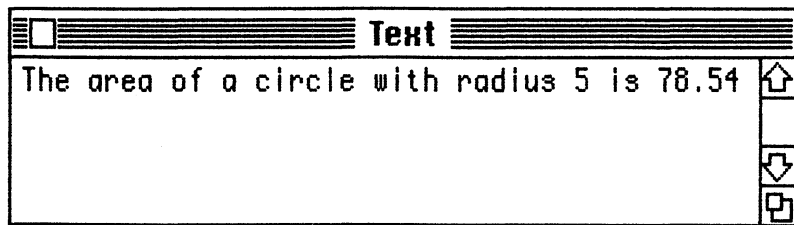


Figure 3-34.

## Read and Readln

It would be nice if we could assign a value to Radius as the program is running. There are two statements that allow us to do this in Pascal, Read and Readln.

The Readln statement stops and waits for information to be entered from the keyboard. The information that is entered is displayed in the Text window and put into the variable that is in between the parentheses. The variable must be declared in the var section of the program. The user signals that he has finished the entry of the value by pressing the return key. An example of the program using Readln can be found in Figures 3-35 and 3-36.

The Read statement works in the same fashion as the Readln except that the input does not have to be terminated by hitting the return key. Entry of information is terminated when a space



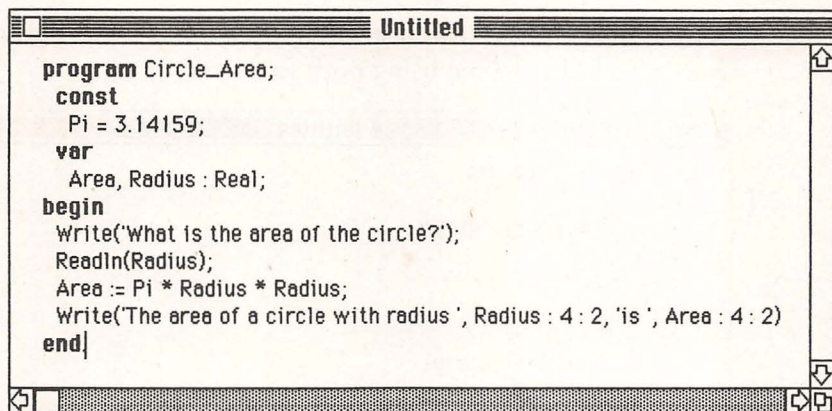


Figure 3-35.

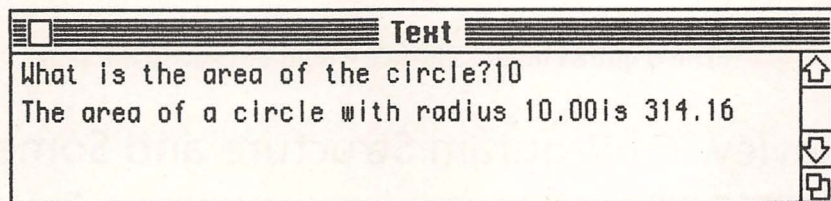


Figure 3-36.

or comma is entered. All of the previously discussed data types can be used with the Read and Readln statement. Read and Readln can also be used with more than one variable. If more than one variable is used, each variable is separated by commas.

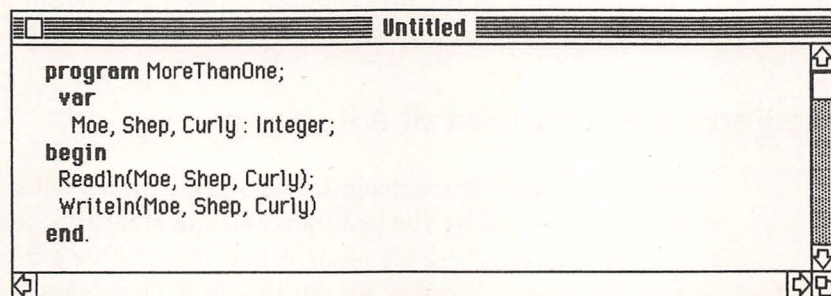


Figure 3-37.

The user must type some non-numeric character between the three integers to let the computer know where one integer ends and the next begins. Because we are using a Readln statement a

return must follow the last value entered. This program could have also been written using both Read and Readln.

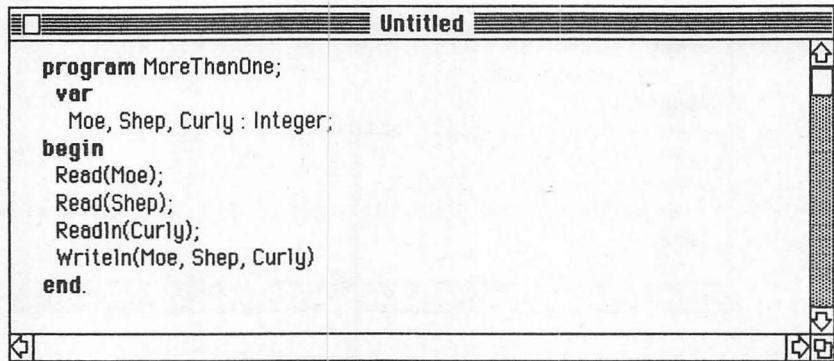


Figure 3-38.

The required input sequence would be exactly the same.

## Review Of Program Structure and Some Examples

A program contains three parts:

1. The heading,
2. The variable declarations,
3. The statements.

Let's now look at the process of designing a program to solve a specific problem.

### The Perimeter and Area of a Rectangle

Before we can write a program we must explore what information will be needed by the program and the algorithm that will be used. An algorithm is a set of steps needed to solve a problem. In this book we will express algorithms in a cross between Pascal and English known as pseudocode.

The pseudocode:

Read the width and length of the rectangle.  
Calculate the area  
Area = Length \* Width



Calculate the perimeter

Perimeter = 2 \* (Length + Width)

Write the information

We are now ready to translate this directly into Pascal.

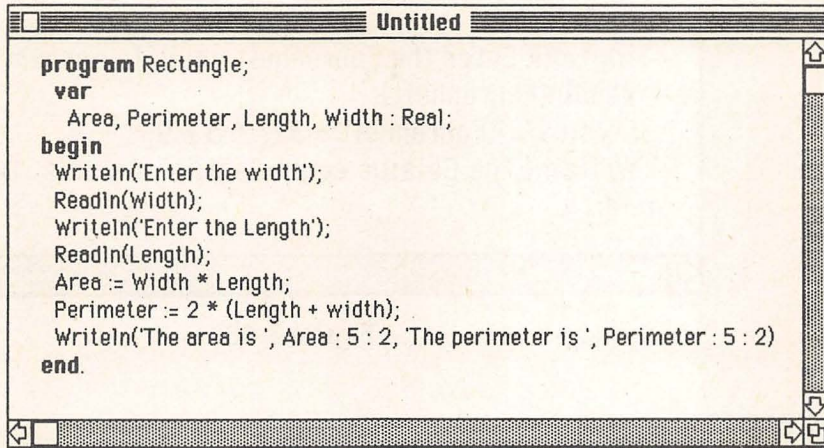


Figure 3-39.

## Converting Temperatures

The next example is a program that converts Fahrenheit temperature to Celsius. The algorithm for this program is:

Read the Fahrenheit value

Calculate Celsius

Celsius = (Fahrenheit - 32) \* 5 / 9

Write result

Notice the programming style utilized in both of the programs (Figure 3-40).

1. The program name reflected the purpose of the program.
2. The variable name conveyed to the reader what it contained.
3. Before any input was required, the user was informed as to what should be entered.

In this chapter, we have discussed the basic elements of a Pascal program. In the next chapter, we will build upon these foundations and begin to develop more sophisticated Pascal programs.

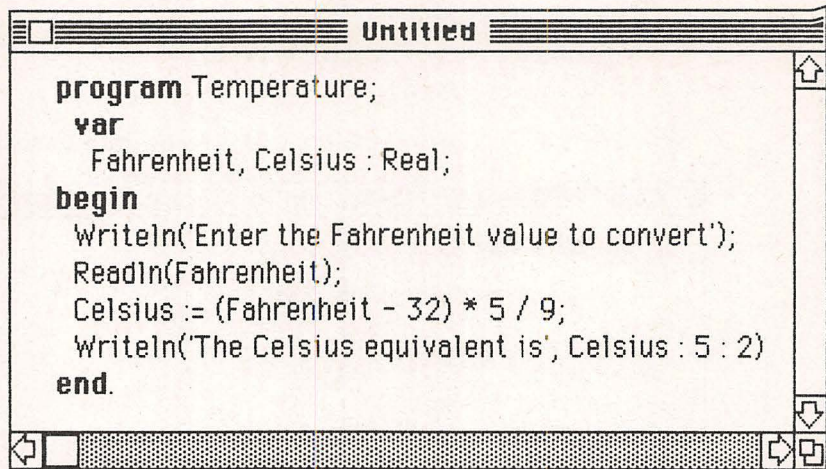


Figure 3-40.

## Exercises

- Which of the following identifiers are illegal and which are legal?
  - Bill
  - 3CP0
  - R2D2
  - NUMB5
  - Program
  - Star\*s
- What is the value of each of the following expressions?
  - $4 + 7 - 13 * 2$
  - $8 \bmod 3 + 1$
  - $4 * 8.2 / 4.1$
  - $4.0 * 3.1 + 18$
  - $(15 + 3) \operatorname{div} 7$
- What are the data types of the expressions in Exercise 2?
- Write a program that accepts two values and averages them.
- Write a program to find the square and cube of a value entered.



# 4 Pascal Structure

We have already seen the basic elements of a Pascal program: how a program is formed; how values are assigned to variables; and how simple input and output is done. However more is needed to write programs that are capable of doing more than just trivial tasks. This chapter will introduce you to program structures, the building blocks of programs. But first we need another one of Pascal's data types.

## The Boolean Data Type

Another of Pascal's data type is Boolean. A variable of type Boolean can have one of only two possible values, TRUE or FALSE. They are called Booleans in honor of George Boole, the father of algebraic logic.

A variable is declared a Boolean with:

```
var  
  X, Y: Boolean;
```

Here the variables X and Y are both of type Boolean. A value is assigned to a Boolean variable with an assignment statement.

```
X := TRUE;  
  or  
Y := FALSE;
```

## Boolean Operators

Since Boolean's are not numeric values, they need a special set of operators to be used with them. The operators are the standard

logical operators **and**, **or** and **not**. **And** and **or** are binary operators taking two Boolean values and giving a single value as a result. **Not** is a unary operator which takes only a single value.

The operator **and** gives a result of TRUE only if both values are TRUE. **Or** gives a result of TRUE if either value is TRUE. The table of possible results of a Boolean operator is called a truth table.

### AND

| <i>Value1</i> | <i>Value2</i> | <i>Result</i> |
|---------------|---------------|---------------|
| TRUE          | TRUE          | TRUE          |
| TRUE          | FALSE         | FALSE         |
| FALSE         | TRUE          | FALSE         |
| FALSE         | FALSE         | FALSE         |

### OR

| <i>Value1</i> | <i>Value2</i> | <i>Result</i> |
|---------------|---------------|---------------|
| TRUE          | TRUE          | TRUE          |
| TRUE          | FALSE         | TRUE          |
| FALSE         | TRUE          | TRUE          |
| FALSE         | FALSE         | FALSE         |

The third Boolean operator **not** simply reverses whatever value it is given. **Not** TRUE is FALSE, **not** FALSE is TRUE.

## Boolean Expressions

Just like numeric expressions where many values are reduced to a single value, there are also Boolean expressions. A Boolean expression can consist of:

1. A single Boolean value.
2. A single Boolean variable.
3. A combination of values and variables connected by operators.

When more than one operator is used in an expression the order of precedence is:



not  
and  
or

This order can be altered with the use of parentheses. It is recommended you use parentheses in large Boolean expressions to make it more readable and understandable.

| Expression                | Result |
|---------------------------|--------|
| TRUE                      | TRUE   |
| TRUE or FALSE             | TRUE   |
| TRUE or FALSE and FALSE   | TRUE   |
| (TRUE or FALSE) and FALSE | FALSE  |
| not TRUE or FALSE         | FALSE  |

Assuming

X := TRUE;

Y := FALSE;

TRUE or Y and not X                      TRUE

Only a Boolean value can be assigned to a Boolean variable. Pascal goes to quite a bit of trouble to check that all values assigned are compatible to the type of variable used. Type checking is done to prevent erroneous values from being assigned to variables. Due to the large amount of type checking done, Pascal is referred to as a strongly typed language.

## IF-THEN

The **if-then** structure allows for execution of different statements depending upon the result of a Boolean expression. The **if** statement is a building block of all programming languages.

The form of the **if** statement is (see Figure 4-1):

```
if Boolean expression is true then
    statement1;
statement2;
```

If the expression has a value of TRUE then Statement1 is executed, otherwise it will be skipped. Statement2 is executed no matter what the value of the expression. The following program

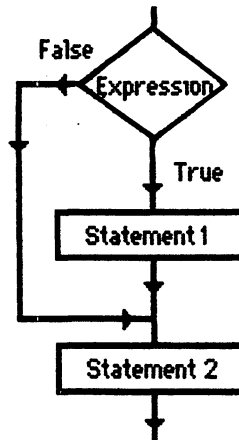


Figure 4-1.

segment uses an if-then statement to test the value of a number that is entered.

```
Readln(Num);  
Write('A ');  
if Num > 100 then  
    Write('BIG');  
Write('NUMBER');
```

If the number entered is less than or equal to 100 then the output is:

**A NUMBER**

If the number entered is greater than 100, the Write statement after the THEN is executed and the output is:

**A BIG NUMBER**

The most common Boolean expression used in an if statement is a conditional expression also known as a condition test. A condition expression is a comparison of two values and results in a Boolean value of TRUE or FALSE. In the if statement above the expression `Num > 100` is a conditional expression. The following are the possible conditions that can be tested:

|     |              |
|-----|--------------|
| =   | Equal to     |
| < > | Not equal to |
| <   | Less than    |
| >   | Greater than |



$> =$  Greater than or equal to  
 $< =$  Less than or equal to

Here are some examples of conditional expressions and their values. Assume I has a value of 3 and J has a value of 4.

| Expression    | Value |
|---------------|-------|
| $3 > 2$       | TRUE  |
| $I * 2 = 4$   | FALSE |
| $I < > J$     | TRUE  |
| $I - J < = J$ | TRUE  |

Here is the syntax of the if-then statement.

IF THEN STATEMENT

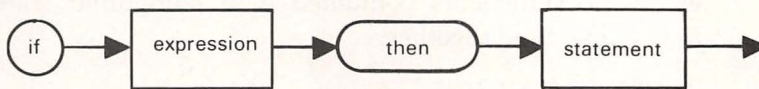


Figure 4.2. Syntax Diagram : IF-THEN Statement.

The following program reads three positive values from the keyboard and finds the largest of the three.

```

program lffy;
  var
    Num1, Num2, Num3 , Biggest: Integer;
begin
  Writeln('Enter 3 numbers');
  Readln(Num1, Num2, Num3);
  Biggest := 0;
  if Num1 > Biggest then
    Biggest := Num1;
  if Num2 > Biggest then
    Biggest := Num2;
  if Num3 > Biggest then
    Biggest := Num3;
  Write('The Largest Value was',Biggest)
end.

```

In this program each value is compared against the current largest value to see if it is larger. The variable representing the largest value, Biggest, is initialized to zero which is smaller than

any value that will be entered. This assures that the first meaningful value of `Biggest` will be the value of `Num1`.

## Compound Statements

If we were only allowed to use a single statement as part of an **if-then** statement, its usefulness would be severely limited. If the condition is **TRUE** there exists a way to execute many statements rather than just one. This involves re-examining the definition of a statement. So far, the term statement has been used to mean a single statement followed by a semicolon. But wherever a single statement can be used, it can be replaced by a compound statement. A compound statement is a sequence of one or more statements separated by semicolons and bracketed by a **begin** and **end**. The statements contained in a compound statement are always executed together.

COMPOUND STATEMENT

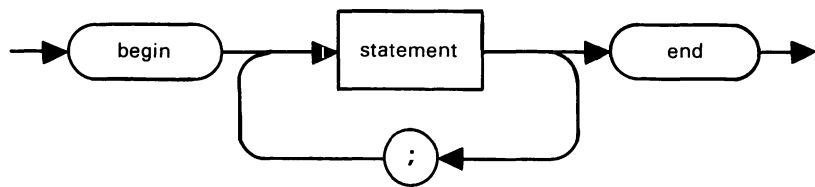


Figure 4-3. Syntax Diagram: Compound Statement.

This second use of **begin** and **end** to bracket compound statements might seem a little confusing. However, **begin** and **end** are not limited to just marking the start and finish of the statements in a program. A closer examination of the definition of a program shows that the **begin** and **end** are really used for the same purpose; to bracket a group of statements always done together, that is, the statements in a program. The following program uses compound statements to execute more than one statement as part of an **if-then**. Notice the syntax. A statement before an **end** never gets a semicolon. An **end** which comes before another statement gets a semicolon.

```

program CompoundExample;
  var
    Num : Integer;

```



```

begin {Program}
  Writeln('Enter number');
  Readln(Num);
  if Num < 0 then
    begin {Start of Compound statement}
      Num := Num * -1;
      Writeln(' The absolute value is', Num)
    end; {End of Compound statement}
  if Num >= 0 then
    Writeln(' The absolute value is', Num)
end. {Program}

```

This program finds the absolute value of a number which is a number without a sign. If the value that is entered is a negative value we remove the sign by multiplying by -1 and then displaying the number. If the number is positive the number is just displayed as is. This example also demonstrates one of the rules of good programming. Whenever a value is to be entered, the program should prompt the user with a message telling them what is expected.

## IF-THEN-ELSE

The **if-then** statement executes a statement (or compound statement) if the condition evaluated is TRUE. A second clause called **ELSE** allows a second statement (or compound statement) to be executed if the condition evaluated is FALSE. Thus, with **if then else** we can have two mutually exclusive statements (or compound statements), one executed if the condition is true, the other executed if the condition is false.

The form of the IF THEN ELSE is:

```

if condition then
  Statement1
else
  Statement2;
  Statement3;

```

It can be graphically represented with the flowchart in Figure 4-4.

The syntax of the **if then else** varies slightly from the **if then**. Note that there is no semicolon after the **then** statement (Figure 4-5).

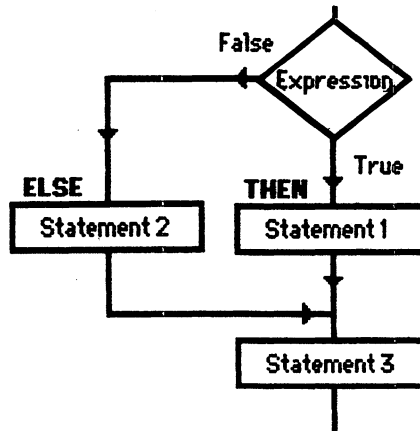


Figure 4-4.

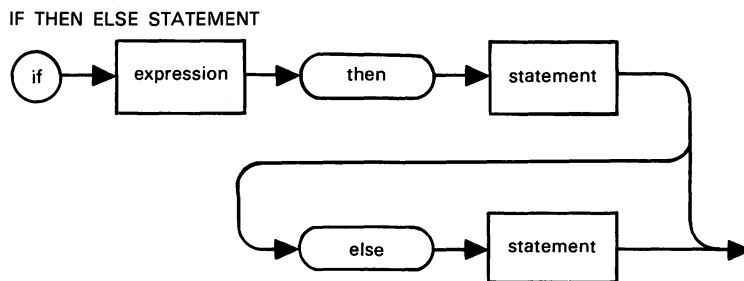


Figure 4-5. Syntax Diagram : if-then-else Statement.

For example:

```

if Hrs ≤ 40.0 then
    Pay := Hrs * Rate
else
    Pay := Hrs * Rate + (Hrs - 40) * Rate * 1.5;
  
```

In this example from a hypothetical payroll program, the if statement is used to determine the pay based on the number of hours worked. If the value of Hrs is 40 or less the statement after the **then** `Pay := Hrs * Rate;` is executed. If the value of Hrs is greater than 40 then the **else** clause `Pay := Hrs * Rate + (Hrs - 40) * Rate * 1.5;` is executed.

The use of **else** can replace the need for multiple if **then** statements. The "find the largest number" program can be rewritten to reflect this.



```

program lffy;
  var
    Num1, Num2, Num3 , Biggest: Integer;
begin
  Writeln('Enter 3 numbers');
  Readln(Num1, Num2, Num3);
  Biggest:=Num1;
  if Num2 > Biggest then
    Biggest:= Num2
  else
    if Num3 > Biggest then
      Biggest:=Num3;
  Write('The Largest Value was',Biggest)
end.

```

## Nested IF Statements

The statement that follows after a **then** or **else** can also be an **if** statement. The nesting of **if** statements can be used to make multiple decisions based on the same data. The following program makes use of nested **if** statements.

```

program Swimming;
  var
    Temp : Integer;
begin
  Writeln('WHATS TODAYS TEMPERATURE?');
  Readln(Temp);
  if Temp > 70 then
    if Temp > 80 then
      Write('GO TO THE BEACH')
    else
      Write('GO TO THE POOL')
  else
    Write('GO TO THE MOVIES');
end.

```

Here is a list of possible inputs and their associated output.

| Input | Output           |
|-------|------------------|
| 75    | GO TO THE POOL   |
| 60    | GO TO THE MOVIES |
| 89    | GO TO THE BEACH  |

In the program two **else** statements are used. An **else** always belongs to the **if-then** that is closest to it. In order to see what belongs to what, it is important that a program be indented properly. The reader should immediately be able to tell which **else** belongs to which **if** based upon the way it is indented. Fortunately, MacPascal automatically indents a program for you as you enter it. Here is a look at another way of structuring the **if** statements.

```
if Temp > 80 then
  Write('GO TO THE BEACH')
else
  if Temp > 70 then
    Write('GO TO THE POOL')
  else
    Write('GO TO THE MOVIES');
```

## FOR Loops

Up to this point all the programs we have written have one thing in common. They executed in a sequential order, starting with the first statement and proceeding to the last statement (although sometimes an **if** statement provided a fork in the road.) There was no way to repeat the execution of some part of the program. Loops are used for this purpose. Pascal has three different loop structures, **for**, **while**, and **repeat**. Each of the three different loops have their own uses and attributes. The first loop to be discussed is the **for** loop.

The **for** loop is used to repeat the same statement (or compound statement) a specified number of times.

```
for variable := expression to expression do
  statement;
```

A **for** loop contains a variable known as the control variable. The initial and final values for the control variable are given as expressions. The control variable is assigned the initial value and it is checked to see if it is less than or equal to the final value. If it is, the statement is executed and the control variable is incremented. The process is then done again.

Here is an example.

```
for K := 1 to 5 do
  Writeln('The value of K is ',K);
```



In the above FOR loop, the statement after the **do** is executed once for every integer value control variable assigned. What is printed is :

The value of K is 1  
 The value of K is 2  
 The value of K is 3  
 The value of K is 4  
 The value of K is 5

The value of the control variable K is changed after each execution of the **Writeln** statement contained in the loop. When the final value of 5 was reached the **Writeln** was executed for the last time.

In the next example, the values of the control variable in the FOR loop are added together.

```
Sum := 0;
for I:= 1 to 3 do
  Sum:=Sum+I;
```

In the above for loop, the statement after the **do** is executed once for every integer value of the control variable (I) from one to three.

Let's trace the execution of the loop presented in the example

| I | Sum |                    |
|---|-----|--------------------|
| - | 0   | (before execution) |
| 1 | 1   |                    |
| 2 | 3   |                    |
| 3 | 6   |                    |

You can see that the loop was executed three times, once for each value of I from 1 to 3. This loop does the same as the following statements.

```
Sum := Sum + 1;
Sum := Sum + 2;
Sum := Sum + 3;
```

The statement in the loop accomplished all three of these additions. The value added to Sum is the same as the control variable which is changed in the loop.

Here is the syntax diagram for the for loop.

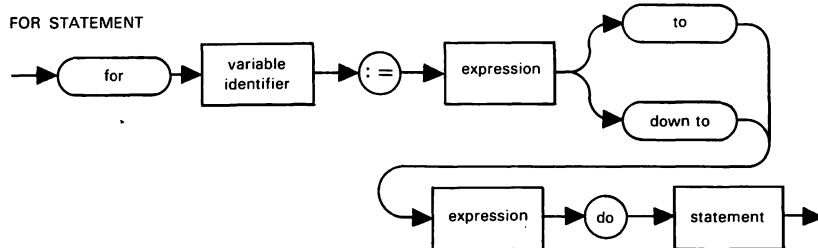


Figure 4-6. Syntax Diagram : FOR Statement.

The statement in a for loop is executed once for every integer value from the first value given until the last value given. The final value must be greater than the initial value. If it is not, the statement in the loop will not be executed. The control variable must be an integer or, as we will see later, any ordinal type. The value of the control value can not be changed inside the loop. An attempt to do so will produce an error. Once the loop has terminated, the value of the control variable is said to be undefined, meaning its value is not predictable and should not be used.

The statement contained in a for loop can be a compound statement as demonstrated in the next example.

```
Sum := 0;
for I := 1 to 3 do
  begin
    Sum := Sum + I;
    Writeln(Sum)
  end;
Writeln(Sum);
```

The output from this program segment is:

```
1
3
6
6
```

Notice that the first Writeln statement was in the loop and thus executed three times printing the 1, the 3, and the first 6. The second Writeln statement which printed the second 6 was outside the loop and thus executed only once after the loop terminated.

The for loop is used in situations where the number of iterations is known or can be determined prior to executing the loop.



For example, the following program will read five integer values and print their average.

```
program Average1;
var
    Sum, Count, Num : Integer;
    Avg : Real;
begin
    Sum := 0;
    for Count := 1 to 5 do
        begin
            Writeln('Enter a number');
            Readln(Num);
            Sum := Sum + Num
        end;
    Avg := Sum / 5;
    Writeln('The average is ', Avg : 6 : 2)
end.
```

The for loop was used to read the five values and add them to the variable Sum. Sum was then divided by five to get the average. Notice that all the values were read into the same variable, Num, erasing any previous value it contained. This was fine since once the value was added to Sum, it was no longer needed. This method would not be sufficient if we wanted to retain all five values entered for further use.

In this example, it was known beforehand that only five values would be entered and the program was written accordingly. However, we need not limit ourselves to deciding that when writing the program. We could easily adapt the program to allow us to first enter how many numbers are to be averaged.

```
program Average2;
var
    N, Sum, Count, Num : Integer;
    Avg : Real;
begin
    Sum := 0;
    Writeln('Average how many numbers?');
    Readln(N);
    for Count := 1 to N do
```

```
begin
  Writeln('Enter next number to average');
  Read(Num);
  Sum := Sum + Num
end;
Avg := Sum / N;
Writeln('The average is ', Avg : 6 : 2)
end.
```

The number of values to be averaged is entered first and used as the final value of the **for** loop. It is also used as the divisor in the statement that calculates the average.

We can repeat the entire sequence of instructions in this program by placing the **for** loop in a second **for** loop.

```
program Average3;
var
  Times, N, Sum, Count, Num : Integer;
  Avg : Real;
begin {Program}
  for Times := 1 to 3 do
    begin {FOR Times Loop}
      Sum := 0;
      Writeln('Average how many numbers?');
      Readln(N);
      for Count := 1 to N do
        begin
          Writeln('Enter next number to average');
          Readln(Num);
          Sum := Sum + Num
        end;
      Avg := Sum / N;
      Writeln('The average is ', Avg : 6 : 2)
    end {FOR Times Loop}
  end. {Program}
```

In this new version of the average program, a second **for** loop (**for** Times) has been added. For each value of Times (from 1 to 3) the entire process is done. This is known as nested **for** loops. The inside loop is done completely for each value of the outer loop. Nested **for** loops will be discussed again when we talk about arrays.



## DOWNTO

There is a slight variation in the **for** loop that allows the loop to count down rather than up.

```
for I := 5 downto 1 do  
    Write(I);
```

This loop prints 5 4 3 2 1. The keyword **downto** has replaced **to** indicating the direction of the counting. The initial value must now be greater than the final value or the statement will not be executed.

One limitation of the **for** loop is that the control variable can only be increased or decreased by one. This can be overcome by using a second variable that is independent of the control variable. Let's write a loop to add the even numbers between 1 and 10.

```
EvenNumber := 0;  
EvenSum := 0;  
for Count := 1 to 5 do  
    begin  
        EvenNumber := EvenNumber + 2;  
        EvenSum := EvenSum + EvenNumber  
    end;  
    WriteLn(EvenSum);
```

Notice that Count goes from 1 to 5 but at the same time the values of EvenNumber are from 2 to 10 by 2s.

## Calculating Interest Compounded Daily

A **for** loop can be used in a program to calculate how an amount of money will increase when interest compounded daily is added to it for any period of time. In the following program the user is asked to enter the principal amount and the number of years for which to compute the interest. The interest is compounded daily and added to the principal. The information is printed every 30 days. The formula for one day's interest is:

$$\text{Interest} := \text{Principal} * (\text{Rate} / 365)$$

Here is the pseudocode for the program.

Get principal amount  
Get number of year  
Get interest rate  
for each year do  
    for every day in the year do  
        calculate the daily interest  
        add it to the principal  
        if the day is divisible by 30 then Write (information)

Now here is the program.

```
program Interest;
const
    DaysInYear = 365
var
    Day, Years, Yr : Integer;
    Rate, Interest, Principal : Real;
begin
    Writeln('Enter the principal');
    Readln(Principal);
    Writeln('Enter the number of years');
    Readln(Years);
    Writeln('Enter the interest rate as a percent');
    Readln(Rate);
    Rate := Rate / 100; {Convert rate to a fraction}
    for Yr := 1 to Years do
        for Day := 1 to 365 do
            begin
                Interest := Principal * (Rate / 365); {Calculate
                    daily interest}
                Principal := Principal + Interest; {Add interest
                    to principal}
                if Day mod 30 = 0 then
                    Writeln('For day', Day, 'The New Principal is',
                        Principal)
            end {FOR loop}
        end. {Program}
```

For the values of Principal equal to 1000, number of years equal to 1 and interest rate equal to 8% the output would be:

In year 1 day 30 The new principal is 1006.60  
In year 1 day 60 The new principal is 1013.24  
In year 1 day 90 The new principal is 1019.92



In year 1 day 120 The new principal is 1026.65  
 In year 1 day 150 The new principal is 1033.42  
 In year 1 day 180 The new principal is 1040.24  
 In year 1 day 210 The new principal is 1047.10  
 In year 1 day 240 The new principal is 1054.01  
 In year 1 day 270 The new principal is 1060.96  
 In year 1 day 300 The new principal is 1067.96  
 In year 1 day 330 The new principal is 1075.00  
 In year 1 day 360 The new principal is 1082.09

## The Summation of an Infinite Series

Mathematicians tell us the sum of the infinite series  $1 + 1/2 + 1/4 + 1/8 + 1/16 + 1/32 \dots$  is 2. This was proved before the invention of computers through mathematical induction. We can use MacPascal to prove this with the use of a **for** loop. The loop will successively add together the terms of the series, and print both the term and the current sum. This example is limited by the accuracy of the real data type which is 7 or 8 decimal places. Once that is exceeded values are rounded off.

```

Set first term to 1
for I := 1 to 30 do
  add the term to the sum
  divide the term in half to get the next term
  write the information

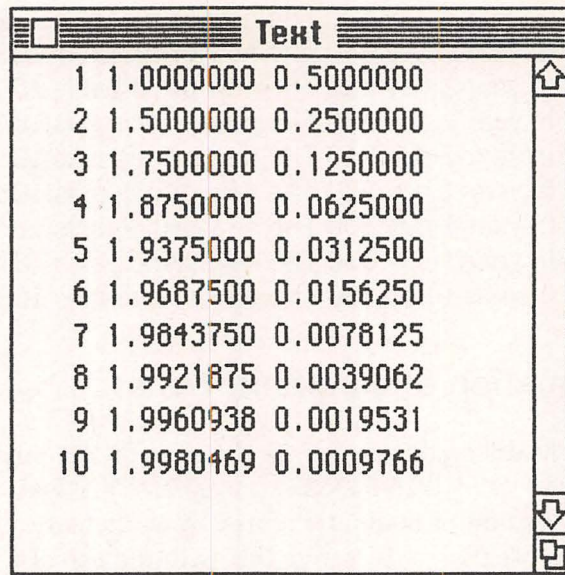
```

```

program Prove__An__Old__Theory
var
  I : Integer;
  Sum, Term : Real
begin
  Term := 1;
  Sum := 0;
  for I := 1 to 30 do
    begin
      Sum := Sum + Term;
      Term := Term / 2;
      Writeln(I, Sum : 9 : 7, Term : 9 : 7)
    end
  end.

```

The output of the program looks like this in the Text window.



|    |           |           |
|----|-----------|-----------|
| 1  | 1.0000000 | 0.5000000 |
| 2  | 1.5000000 | 0.2500000 |
| 3  | 1.7500000 | 0.1250000 |
| 4  | 1.8750000 | 0.0625000 |
| 5  | 1.9375000 | 0.0312500 |
| 6  | 1.9687500 | 0.0156250 |
| 7  | 1.9843750 | 0.0078125 |
| 8  | 1.9921875 | 0.0039062 |
| 9  | 1.9960938 | 0.0019531 |
| 10 | 1.9980469 | 0.0009766 |

Figure 4-7.

Experiment by changing the number of iterations and the field width parameters.

## The WHILE Loop

The second of the Pascal loop structures is the **while** loop. Unlike the **for** loop, where the number of iterations is determined before the loop is executed, the **while** loop is a free loop where the number of iterations is dependent on what happens inside the loop body.

The structure of the **while** loop is:

```
while Boolean expression is true do  
    Statement1;  
    Statement2;
```

First, the value of the Boolean expression is evaluated (similar to the **if** statement), if its value is true the statement (or compound statement) which forms the loop body is executed. Then the Boolean expression is once again checked. This process continues until the value of the Boolean expression becomes false. The operation of the **while** loop is indicated in Figure 4-8.

The syntax of the **while** loop is described in Figure 4-9.



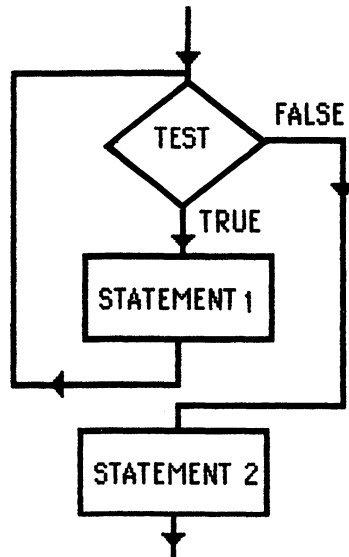


Figure 4-8.

WHILE STATEMENT

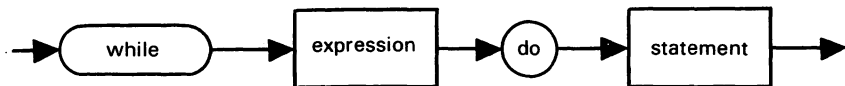


Figure 4-9. Syntax Diagram : WHILE Statement.

The following program segment contains a **while** loop that prints the integers from 1 to 5.

```

Int := 1;
while Int <= 5 do
  begin
    Writeln(Int);
    Int := Int + 1
  end;

```

The output from this loop is :

```

1
2
3
4
5

```

Notice that this loop contains a compound statement. Since the condition has to be changed in the loop body almost all while loops will contain a compound statement. The body of the loop was executed 5 times. Each execution of the body writes the value of `Int` and then adds one to it. The last iteration writes the value 5 and then increments `Int` to 6. This causes the condition to become `False` and the loop to terminate. A `Writeln` statement placed outside the loop will show the value of `Int` to be 6. When using `while` loops it is *imperative* that the condition checked is eventually changed inside the loop body. The following loop is an infinite loop, one that will never terminate since the condition is never changed.

```
Int := 1;
while Int <= 10 do
  Writeln(Int);
  Int := Int + 1;
```

This looks the same as the one shown before, but a closer look unveils a difference. The statement contained in this loop is a single statement (`Writeln`), not a compound statement. Therefore, the variable `Int` will always contain a value of 1 and never change since the statement which increments it is outside the loop. Since `Int` is never incremented, the `Writeln` statement is executed over and over again, infinitely.

In the next example, a `while` loop is used to simulate the `div` and `mod` operators.

```
program Divide;
var
  Top, SaveTop, Bottom, Answer, Remainder : Integer;
begin
  Writeln('Enter the dividend and divisor');
  Readln(Top, Bottom);
  Answer := 0;
  SaveTop := Top;
  while Top >= Bottom do
    begin
      Top := Top - Bottom;
      Answer := Answer + 1
    end; {while}
```



```

    Remainder := Top;
    Writeln(SaveTop : 1, '/' ,Bottom : 1, '=' ,Answer : 1, 'R',
    Remainder : 1)
end. {Program}

```

Trace the program for the values Top = 18 and Bottom = 5.

| Top             | Bottom | Answer | Remainder | SaveTop |
|-----------------|--------|--------|-----------|---------|
| 18              | 5      | 0      | -         | 18      |
| 13              | 5      | 1      | -         | 18      |
| 8               | 5      | 2      | -         | 18      |
| 3               | 5      | 3      | -         | 18      |
| loop terminates |        |        |           |         |
| 3               | 5      | 3      | 3         | 18      |

Notice that SaveTop was needed to hold the value of Top since the original value is changed in the loop.

## Sentinels

A sentinel is a technique used to signal the end of a stream of input. To demonstrate let's rewrite the averaging program with a **while** loop. The loop will keep reading data until the user enters a -1. This is used to signal that there will be no more input and the average should now be calculated.

```

program Average;
var
    Sum, Num, Count : Integer;
    Average : Real;
begin
    Sum := 0;
    Num := 0;
    Count := 0;
    Writeln('Enter number to average');
    Readln(Num);
    while Num < > -1 do
        begin
            Sum := Sum + Num;
            Count := Count + 1;
            Writeln('Enter number to average');
            Readln(Num)
        end; {while}
    Average := Sum / Count;
    Writeln('The average is ', Average : 6 : 2)
end.

```

The variable `Count` is used to keep track of the number of values entered. It is then used to calculate the average.

## The Break Even Point

The following program finds an individual's break even point. Your break even point is that week of the year when you stop earning money for the government, and start earning it for yourself. The program prompts the user to enter his weekly salary and the total of the weekly deduction from his paycheck. The yearly taxes are then calculated and a `while` loop subtracts the weekly salary from that amount until it is paid off.

```
program BreakEven;
var
    WeeklyDeduct, YearlyDeduct, WeeklySalary : Real;
    Week : Integer;
begin
    Writeln('Enter your weekly salary');
    Readln(WeeklySalary);
    Writeln('Enter total of your weekly deductions');
    Readln(WeeklyDeduct);
    YearlyDeduct := WeeklyDeduct * 52;
    Week := 0;
    while YearlyDeduct > 0 do
    begin
        Week := Week + 1;
        YearlyDeduct := YearlyDeduct - WeeklySalary
    end; {While loop}
    Writeln('Your break even point is in week number',
        Week, 'of the year');
    Writeln('Congratulations')
end.
```

For instance, if your salary is \$250 a week and your weekly deductions are \$85, the program would display:

```
Your break even point is in week number 18 of the year
Congratulations
```

## Controlling the Text Window

The size of the Text window can be controlled from inside a MacPascal program. In a program you can enlarge the Text win-



dow to occupy the entire screen, hiding the other windows. You may find this desirable in a program that is to be used by someone not familiar with the MacPascal environment. Controlling the Text window is done with the help of the Macintosh's Toolbox of commands which are to manipulate the various resources of the computer. These commands are being presented for use in your programs as is. They will be explained in detail in Chapter 10, which deals with graphics.

## The Macintosh Screen

MacPascal is capable of displaying high quality graphics and animation in the **Drawing** window. Before we can write graphics programs we must first take a look at the Macintosh's graphics coordinate system.

The Macintosh's screen can be thought of as a grid of 512 vertical lines and 342 horizontal lines not much different from graph paper. The vertical lines (X coordinates) are numbered from 0 to 511 and the horizontal lines (Y coordinates) are numbered from 0 to 341. The position where a horizontal line and a vertical line intersect is called a point and noted as (X, Y). The upper left corner of the screen, the origin, is where vertical line 0 and horizontal line 0 intersect, point (0,0). The lower right corner of the screen is where vertical line 511 intersects horizontal line 341, point (511,341). Below and to the right of each point on the screen is a dot that can be displayed as either white or black. These dots are called picture elements or pixels for short. For each of the 175,104 ( $342 \times 512$ ) points on the screen is a corresponding pixel (Figure 4-10).

The coordinate system actually extends beyond what is visible on the screen. For instance, the coordinate (-10, -10) is above and to the right of the origin in the upper left hand corner of the screen. These points exist to help in calculating complex geographic constructs which may extend beyond the visible portion of the screen.

To enlarge the Text window to occupy the entire screen add the following commands to your program.

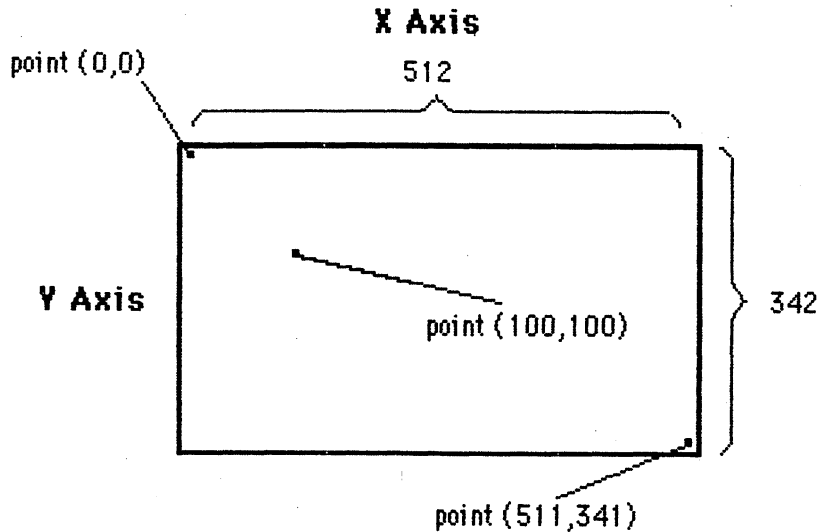


Figure 4-10.

```

var
    TextRect : Rect;
:
begin
    SetRect(TextRect, 0, 0, 511, 341);
    SetTextRect(TextRect);
    ShowText;
:

```

Three commands have to be included in the program as well as the addition of a variable of type Rect (used in graphics operations, see Chapter 10). This will display the Text window over the entire screen, even the Title Bar will be hidden from view. If you want the Title Bar to show simply substitute this command for the other SetRect:

```
SetRect(TextRect, 0, 35, 511, 341)
```

A hidden window can always be revealed by selecting its name from the **Windows** menu.

In subsequent chapters, you will be introduced to other Toolbox commands that will enhance your programs by giving you more control over the environment they are running in.

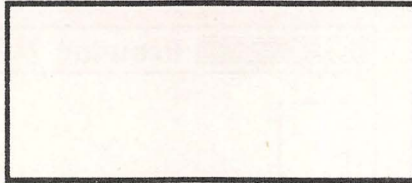


## Introduction to Graphics

The Quickdraw library has many built-in commands to display graphics in the **Drawing** window. The first set of commands we will look at are the rectangle commands.

A rectangle is defined in terms of two points, the point in the upper left hand corner and the point in the lower right hand corner.

(Upper, Left)



(Lower, Right)

Figure 4-11.

To display a rectangle first you must declare a variable to be of the special Quickdraw data type called Rect. This is not a standard Pascal data type and will only be used to display rectangles.

This variable declaration :

**var**

Square, Oblong : Rect;

declares the variables Square and Oblong to be of the special Quickdraw type of Rect.

A variable of type rectangle is used to hold the two points that define a rectangle. This is accomplished with the use of the SetRect command.

```
SetRect(Square, 10, 10, 40, 40);
```

```
SetRect(Oblong, 50, 50, 80, 90);
```

The first command defines Square as having an upper left hand corner of 10,10 and the lower right hand corner of 40, 40. The second command defines Oblong as having an upper left hand corner of 50, 50 and the lower right hand corner of 80, 90. We can now display the two rectangles in the **Drawing** window with the FrameRectangle command.

```
program DrawRectangles;  
  var  
    Square, Oblong : Rect;  
begin  
  SetRect(Square, 10, 10, 40, 40);  
  SetRect(Oblong, 50, 50, 80, 90);  
  FrameRect(Oblong);  
  FrameRect(Square)  
end.
```

Displayed in the window is :

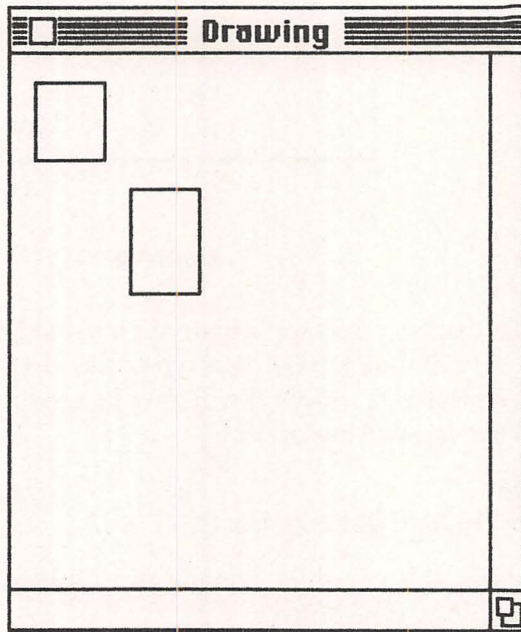


Figure 4-12.

Now that we know how to draw a single rectangle we can create interesting effects by using a `for` loop to display a series of overlapping rectangles. We simply have to continually shift the rectangle we define a slight bit and then display it.

```
program OverlappingRectangles;  
  var  
    Square : Rect;  
    Shift, I : Integer;
```



```
begin
  Shift := 0;
  for I := 1 to 20 do
    begin
      SetRect(Square, 10 + Shift, 10 + Shift, 40 + Shift,
              40 + Shift);
      FrameRect(Square); {Draw Rectangle}
      Shift := Shift + 5
    end
  end.
```

Each point used to define a rectangle in the SetRect command is offset by the value of Shift which is increased in each iteration of the loop. Displayed by the program is:

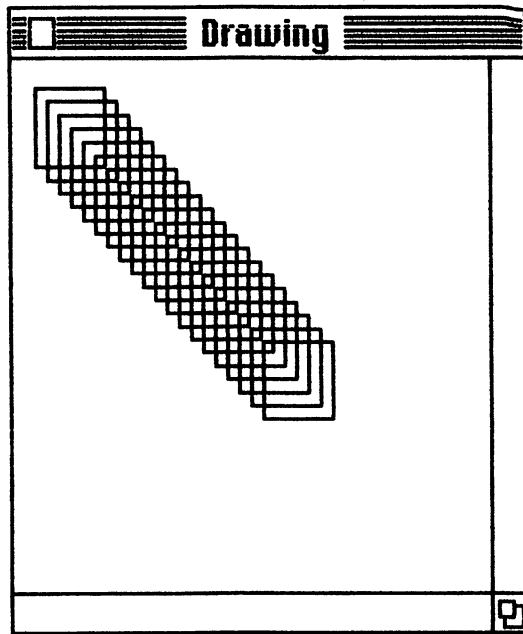


Figure 4-13.

You can vary the number of rectangles and their position by changing the final value of the for loop and the value of Shift.

An animation effect can be achieved by erasing a rectangle after it has been drawn and then redisplaying it shifted slightly. The EraseRectangle command is used to erase the rectangle indicated.

```
program MovingRectangles;
var
  Square : Rect;
  Shift, I : Integer;
begin
  Shift := 0;
  for I := 1 to 20 do
    begin
      Shift := Shift + 5;
      SetRect(Square, 10 + Shift, 10 + Shift, 40 + Shift,
              40 + Shift);
      FrameRect(Square); {Draw rectangle}
      EraseRect(Square) {Erase rectangle}
    end
  end.
```

This program moves the rectangle across the window at a fast speed. You can slow down the animation by wasting time between the writing and erasing of the rectangle. This can be done by inserting a FOR loop that does nothing such as:

```
for K := 1 to 20 do;
```

Notice that no statement is actually executed by the for loop.

Experiment with these programs and try to create interesting displays. You might want to use the PaintRect command instead of FrameRect. PaintRect will display a rectangle that is filled in with black.

It should be self-evident to you that the coordinate system in the Drawing window has 0,0 as the upper left hand point. This will not change even if you enlarge or move the window. The ShowDrawing command placed as the first statement in your program will automatically open the Drawing window from inside your program.

## Exercises

1. Given the following Boolean values for A, B, C, and D, evaluate these Boolean expressions.

A := True; B := False; C := True; D := False

- a. (A and B) or C
- b. D and not(A or (B and C))



- c. A and B or C and D
- d. (A and B) or (C and D)
- e. A and (B or C) and D

2. Determine if the following expressions are True or False.

$X := 3; Y := 5; Z := 14;$

- a.  $(3 > X) \text{ and } (Y < Z)$
- b.  $(X > Y) \text{ or } (Y < X)$
- c.  $(X < Y) \text{ or } (Y > X)$
- d.  $(X < Y) \text{ and } (Y > X)$
- e.  $3 > 2 * X - 15$
- f.  $\text{not}(\text{TRUE}) = (X > Y)$

3. Simplify the following if statements with the use of else.

- a. **if** A > B **then** A := 4;  
   **if** A < B **then** B := 4;
- b. **if** A > B **then** A := 4;  
   **if** A < B **then** B := 4;  
   **if** A = B **then** A := B;

4. How many times would the following for loop statements execute a loop? X and Y are both Integers.

- a. **for** X := 3 **to** 17 **do**
- b. **for** X := 10 **to** 733 **do**
- c. Y := 4;  
   **for** X := Y **to** 18 **do**
- d. **for** X := 4 **to** 4 **do**
- e. **for** X := 41 **downto** 3 **do**
- f. **for** X := 17 **to** 3 **do**

5. What is printed by the following loops.

- a. **for** I := 1 **to** 3 **do**  
   **for** J := 1 **to** 2 **do**  
      Writeln(I, J);
- b. **for** I := 1 **to** 3 **do**  
   **for** J := I **to** 2 **do**  
      Writeln(I, J);

6. What is printed by the following **while** loop.

- a. `l := 4;`  
    **while** `l = 3` **do**  
        `Writeln(l);`
- b. `l := 1;`  
    **while** `l < 10` **do**  
        **begin**  
            `Writeln(l);`  
            `l := l + 1`  
        **end;**
- c. `l := 1;`  
    **while** `l < 10` **do**  
        **begin**  
            `l := l + 1;`  
            `Writeln(l)`  
        **end;**

7. Write a program that will read values from the keyboard and square them. The program should stop when -99 is entered. Display the number and its square in the Text window.

8. Write a program that will display the following.

```
1 = 1
1 * 2 = 2
1 * 2 * 3 = 6
.
.
1 * 2 * 3 * 4 * 5 * 6 = 180
```

9. Write a program that will read from the keyboard an hourly rate of pay and the numbers of hours work. Display this information and the gross pay in the Text window.

10. Alter the program in exercise 9 by calculating time and a half overtime for any hours worked over 40.



# 5 Debugging a MacPascal Program

The cruelest fact of life for programmers is that at least 99% of all programs will not work the first time they are run. Errors, commonly called bugs, can be introduced into the programming process in several places, in typing and in the logic of the actual program. The process of removing errors is called debugging and is a major part of developing a program. Unfortunately debugging more often than not takes longer to do than actually writing the program. Fortunately, MacPascal has several built-in tools to aid in the debugging process. In order to understand more about errors, it is important to first understand the nature of a MacPascal program.

## The Nature of a MacPascal Program

The Run menu (Figure 5-1) contains several options for running a MacPascal program. The process is similar no matter which option is picked. First, the interpreter parses the program for syntax, each statement is examined to see if it conforms to proper Pascal syntax. If a syntax error does exist a dialog box is displayed (see discussion below on syntax errors). At the same time that the syntax is checked, a table is created of all the declared variables and their data types. This table is one of the tools used by the MacPascal interpreter to run a program and keep track of the value of variables. Another important interpreter tool is the program pointer or Finger which points to the next statement in a program to be executed. In some execution modes the Finger can be seen on the screen as a hand with its index finger extended (Figure 5-2).

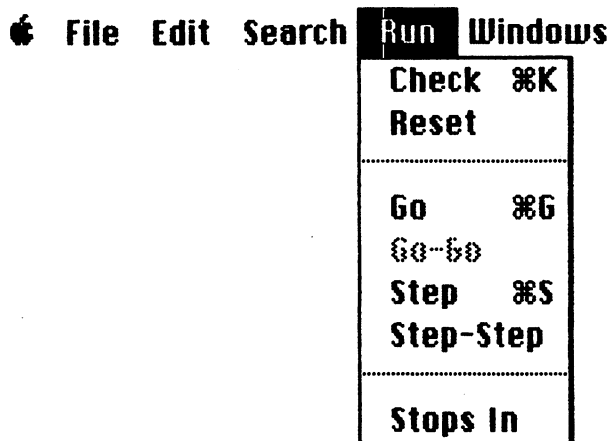


Figure 5-1.

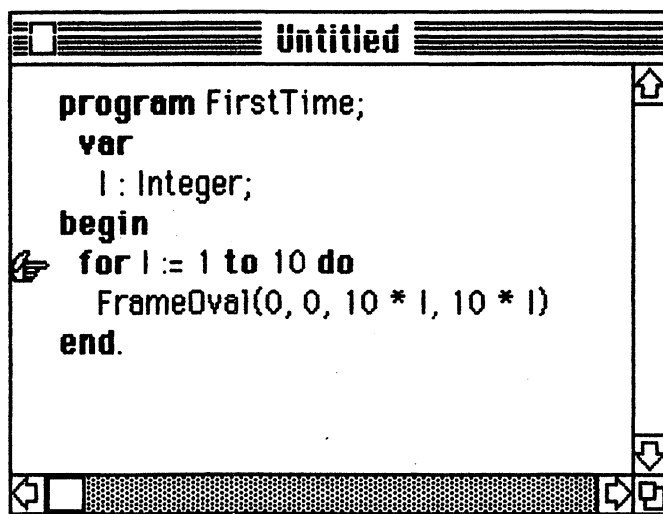


Figure 5-2. The Finger or Program Pointer.

## Execution Modes

**Go ⌘G**

You are already familiar with running a program with Go. The complete program is executed starting with the first statement. When the program is completed the variable table is erased.



**Step** ⌘S

**Step** executes one statement at a time. When using **Step**, MacPascal gives your program the Finger which points to the next line to be executed. This can be done continually to watch the sequence of statements in a program as they execute. **Go** can be selected at any point to start executing automatically from that statement on.

**Step-Step**

**Step-Step** is somewhere between **Step** and **Go**. The statements execute automatically but slow enough that the Finger can be watched.

**Check** ⌘K

**Check** is used to parse the syntax of a program for errors without executing it. You may want to do this occasionally as you enter a large program to expedite the debugging process.

**Pause****Halt**

When a program starts to run, **Pause** appears in boldface in the menu bar. You can use the mouse to select **Halt** from the **Pause** menu. This places the running program into a suspended state with the variables and the position of the Finger still intact. Execution can be restarted from that point by any of the **Run** options.

**Reset**

Whenever a program is halted or paused, the value of the variables remain intact. **Reset** can be used to reset a program to start again from the beginning.

## Syntax Errors

As a program is entered into MacPascal it is automatically indented and the keywords highlighted. The lines entered are also checked for proper syntax. If minor syntax errors occur during typing, the error is indicated by outlining the guilty part of the line. Examples of this type of minor error are:

1. Forgetting to close a string with a quote:

```
Writeln('Good Morning');
```

or



## 2. Mistyping an expression or operator:

$$K = K + 1;$$

Not all syntax errors will be picked up at this stage. Some of them will not be detected until the program is run. Running a program (either with **Go**, **Step** or **Step-Step**) is actually a two step process. The first step is to check the overall syntax of the program, the second step is the actual program execution. During this first step the program is parsed. If a syntax error occurs, the Macintosh beeps and an alert box appears on the top of the screen with a picture of a bug (the type you step on) and description of the error. The box can be cleared by clicking the pointer anywhere in it. The error messages provided are superior to most Pascal systems but sometimes can be cryptic. Turn to Appendix J for a complete listing of the possible syntax errors.

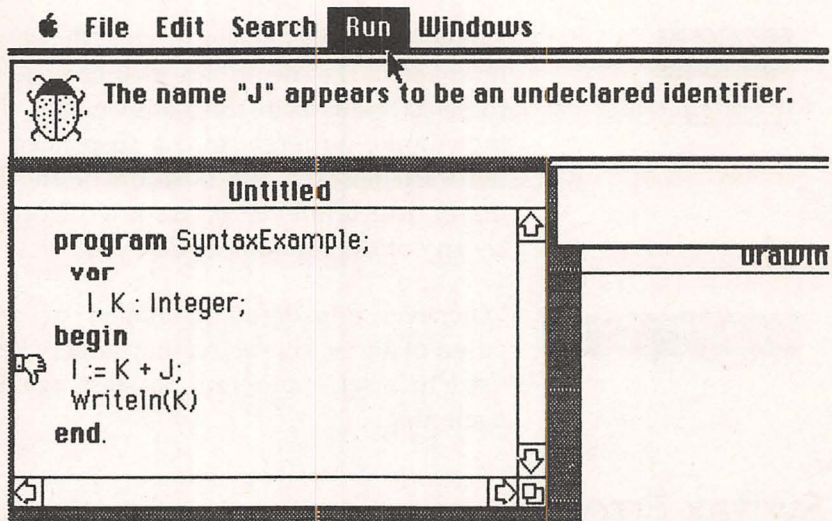


Figure 5-3.

The figure above shows a syntax error in progress. The "Thumbs down" points to the line at which MacPascal realized there was an error. Unfortunately, the line indicated does not always contain the error but rather is the line that triggers the error because of invalid syntax on a previous line. If you can not find the error on the line pointed to, look at the line above it. This will be especially true when the line above is missing a semicolon.

## Undeclared Identifiers

During the checking of a program a list is made of all the variables declared in the `var` section of a program. All variables used in the program are compared against this list. If a variable is not declared an undeclared identifier message is given.

## Execution Errors

Once the syntax of a program is correct it is ready to execute. There are two types of errors that are associated with program execution. Run time errors are errors that will cause a program to stop executing. Logic errors are errors which don't stop a program from running but produce the wrong results. The cause of run time errors are easier to locate than logic errors since MacPascal alerts you to their existence.

### Run Time Errors

A run time error will "bomb" a program that is running, that is, it will stop the execution. An alert box will then appear describing the error. Run time errors occur when an operation that is illegal takes place. An example of a run time error is attempting to divide by zero. Appendix J contains a list of possible run time errors.

### Logic Errors

Once a program is running properly there is no guarantee that the results will be correct. The unwritten law of the programming jungle states that no matter how skillful a programmer, the chance of a logic error increases with the complexity and size of the program. So does the difficulty of locating the source of the trouble. The best way to locate a logic error is to trace through a program making sure that variables have the values you expect and statements are executed in the order anticipated. Fortunately, MacPascal has available debugging tools found in no other Pascal. These tools can greatly ease the debugging process.

## The Observe Window

The **Observe** window is used to view the value of variables or expressions as a program executes. The **Observe** window is opened by selecting **Observe** from the **Windows** menu.

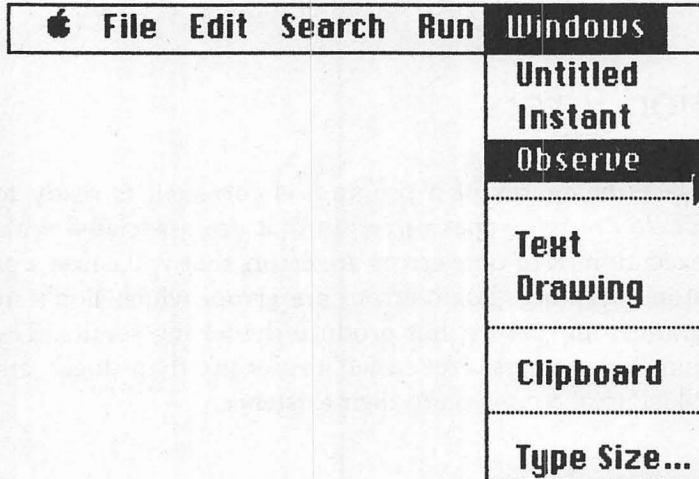


Figure 5-4.

The contents of the window looks like a chart. In the right column you can enter any variable or expression whose value you want to watch by editing it just like you would a program. The number of variables entered can be increased by expanding the size of the window with the size box. This should be done prior to executing the program or during a halt in execution of a program.

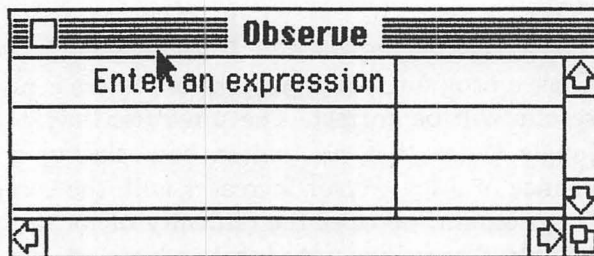


Figure 5-5. The Observe Window.

Variables (or expressions) are entered and edited on the right side of the divider.



The number of variables that can be entered can be increased by enlarging the window with the size box. Once the variables in the window are set the values for the variables are updated every time the program pauses or halts the values. The **Observe** window is especially useful when you **Step** through a program line by line. This allows you to watch how each line in a program affects the value of a variable. **Step-Step**, **Go-Go** or halting the program can also be used.

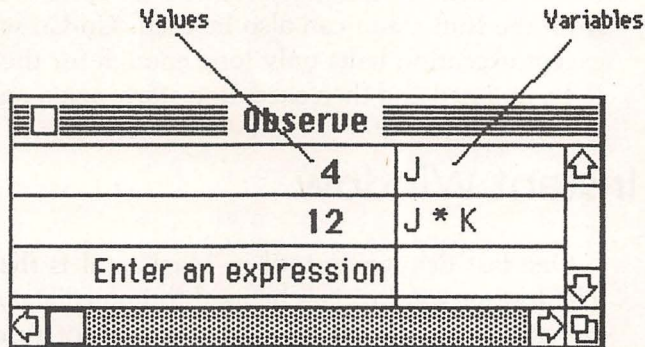


Figure 5-6.

## Setting Break Points

Break points provide a convenient way to halt a running program at a pre-determined line. You could do this using **Step** but you might have to step through a large number of instructions before you get to the spot you want. To set a break point first select **Stops In** from the **Run** menu. The **Program** window will then change in a subtle way. A tiny stop sign will appear in the lower left hand corner of the window and a column will be drawn on the left side of the **Program** window.

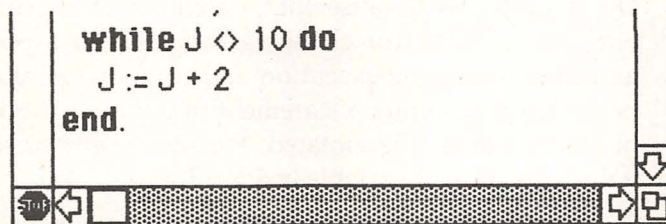


Figure 5-7.

A stop sign can be placed next to any line in the program where you want the program to stop executing. To place a stop sign, move the pointer into the column on the left and watch the pointer change to a stop sign. Position the sign next to the statement and then click the mouse button to set it as a break point. As many break points as you wish may be set. Now every time the program is executed with **Go** or **Step-Step** it will halt at a stop sign. At this point the **Observe** window is updated. Execution can be continued by selecting another **Run** option. The **Go-Go** option from the **Run** menu can also be used. **Go-Go** works just like **Go**, except execution halts only long enough for the **Observe** window to be updated and then execution starts again automatically.

## The Instant Window

One last debugging tool in MacPascal is the **Instant** window. The **Instant** window can be used to immediately execute any Pascal statement any time the program is not running. The **Instant** window is opened by selecting **Instant** from the **Windows** menu.

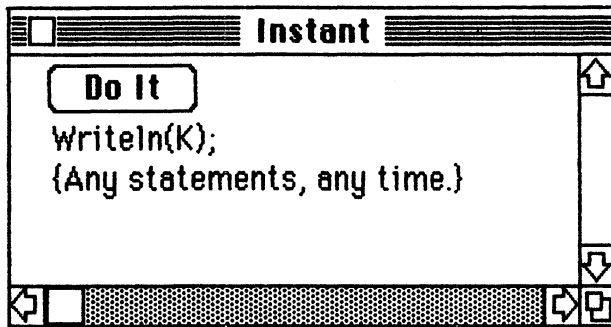


Figure 5-8.

You can then enter and edit any Pascal statement. Clicking the **Do It** button will cause that statement to execute. The **Instant** window is handy for changing the value of a program variable and then re-starting execution and seeing what affect that change had. When executing a statement in the **Instant** window the rules of Pascal can not be violated. For instance, you can't change the value of a control variable inside a **for** loop.

# 6 More On Data Types

In Chapter 3, the data types Integer, Char, and string were introduced. These are not however the only data types available in MacPascal. In this chapter we will take an indepth look at these as well as several other data types in MacPascal and the built-in functions that can be used with them.

## The Char Types

We have seen that the data type string can be used to hold a sequence of characters. The type Char also holds character data but is limited to holding one single character. This may sound strange that both String and the more limited type Char are available. However, historically, standard Pascal only included Char, String is an extension added to the language by many versions of Pascal.

The declaration:

```
var  
  Ch : Char;
```

declares a variable named Ch, capable of holding a single character. The following statements:

```
Ch := 'A';  
Writeln(Ch);
```

assigns a character 'A' to Ch and then displays it.

Character data is stored in the Macintosh in a code known as ASCII (which stands for American Standard Code for Information Interchange). This code uses the numbers 0 through 255 to represent different characters. If we could look into memory



where a character is stored, we would see the ASCII code for that character instead. When character data is stored Pascal remembers to interpret the number as an ASCII code and not an Integer.

## Ordinal Types

Char and most other data types are known as ordinal types. Ordinals are those data types where each possible value (except the last and the first) have a unique predecessor and successor. The ordinal types include, Integer, Boolean, Char and the User-defined types (which are discussed in this chapter). Reals are not an ordinal type since a unique successor and predecessor of a real can not be determined, another decimal place always exists. For example, what real number comes after 5.04, it's not 5.041 because of 5.0401 and 5.04001, this could go on indefinitely. Strings are also not an ordinal type.

A variable of any ordinal type can be used as the control variable in a for loop. This loop:

```
for Ch := 'A' to 'Z' do  
  Writeln(Ch);
```

will iterate 26 times and display all the uppercase letters from A to Z.

## The ORD and CHR Functions

MacPascal has many built-in functions that can be used in programs. A built-in function is similar to an operator, except that it is invoked differently. Like an operator, a function is given a value to work with, called the argument, it can either be in the form of a constant, a variable, or an expression. The function then "returns" a value based on the argument. When used in an expression, functions have the highest order of precedence. That is, functions are evaluated before any other operations take place.

The built-in function ORD takes a character as an argument and returns the ASCII code for that character as an integer.

|          |            |
|----------|------------|
| ORD('A') | returns 65 |
| ORD('a') | returns 97 |
| ORD('B') | returns 66 |

```
ORD('Z')    returns 90
ORD('4')    returns 52
```

The ASCII codes for the characters are in numeric order. This allows for the comparing of two characters to find the alphabetically greater.

```
if 'A' > 'B' then
  Writeln('A is greater')
else
  Writeln('B is greater');
```

This program statement will display : B is greater since the ASCII code for 'B' has a greater numeric value than the ASCII code for 'A'. Try this by typing the if statement into the **Instant** window.

Notice that the letters used in the above example are all uppercase. The lowercase letters have different ASCII codes than the uppercase letters. All the ASCII codes are listed in Appendix L.

A built-in function can be used in almost any instance that a variable can, such as an expression or a Write statement.

```
Writeln(ORD('A'));
X := ORD('A'); { X must be an integer variable}
```

The opposite of the ORD function is the CHR function. CHR takes an integer number, between 0 and 255, interprets it as an ASCII code, and returns the corresponding character. For example:

```
Write(CHR(65));
```

displays an A.

CHR and ORD are inverse functions, thus

```
Write(CHR(ORD('A')));
```

also prints an A. First, the ORD('A') is done returning 65, then CHR(65) is done returning an A.

The ASCII code of the characters that represent digits are also in numeric order.

```
'0' - 48
'1' - 49
'2' - 50
'9' - 57
```

The difference between a single digit integer and its ASCII code may not seem clear to you at this point. The distinction is the set

of operations that can be done on either. You can perform all the arithmetic operations on the integer and only the character operations on the character. We can convert the character representation of a number into that number by subtracting the `ORD('0')` from it.

```
I := ORD('8') - ORD('0')    assigns the integer 8 to I
```

We can use the `ORD` function to display the ASCII values of a set of characters by using a `for` loop.

```
for Ch := 'A' to 'Z' do  
  Writeln(Ch, ORD(Ch));
```

## The SUCC and PRED Functions

The `SUCC` and `PRED` functions will return the successor and predecessor respectively, of any ordinal value.

|                              |              |
|------------------------------|--------------|
| <code>PRED('C')</code>       | returns 'B'  |
| <code>SUCC(50)</code>        | returns 51   |
| <code>SUCC(FALSE)</code>     | returns TRUE |
| <code>SUCC(PRED('B'))</code> | returns 'B'  |

The `PRED` of the first value and the `SUCC` of the last value of an ordinal type is undefined and trying to find it will cause a run time error. For example, `SUCC(TRUE)` doesn't exist and will halt your program.

## Other Built-In Functions

Besides `ORD`, `CHR`, `PRED`, and `SUCC`, there are many other built-in functions that can be used. These functions can be broken up into several categories.

## The Conversion Functions—TRUNC and ROUND

The `TRUNC` and `ROUND` functions are used to convert a Real value into an Integer.

The `ROUND` function rounds off a Real to the closest Integer.



|                              |                                      |
|------------------------------|--------------------------------------|
| <code>I := ROUND(4.3)</code> | assigns the value 4 to the Integer I |
| <code>ROUND(3.002)</code>    | returns 3                            |
| <code>ROUND(3.75)</code>     | returns 4                            |
| <code>ROUND(20.5)</code>     | returns 21                           |
| <code>ROUND(-20.4)</code>    | returns -20                          |
| <code>ROUND(-20.6)</code>    | returns -21                          |

The TRUNC function, which stands for truncation, converts a Real value to an Integer by cutting off the fractional part of the number. The number is not rounded.

|                           |             |
|---------------------------|-------------|
| <code>TRUNC(4.3)</code>   | returns 4   |
| <code>TRUNC(20.7)</code>  | returns 20  |
| <code>TRUNC(-20.7)</code> | returns -20 |

## More On Reals and Integers

Pascal is highly regarded as a programming language to express ideas and to teach programming practices. The major complaint has been that most versions of Pascal did not provide the great accuracy needed in statistical or scientific work or the freedom from rounding errors needed in banking and business work. Thus, old fashioned languages such as FORTRAN and COBOL are still in widespread use. MacPascal has solved these problems by providing easy to use extensions to the Real and Integer data types that provide precision unheard of before in any microcomputer or even minicomputers.

### The LongInt Data Type

The range of values for an Integer is from -MaxInt to MaxInt or from -32767 to 32767 ( $-2^{15}-1$  to  $2^{15}-1$ ). In circumstances where an Integer is needed that will exceed this range, the type LongInt (for long Integer) is available. The range of values for LongInt are -2,147,482,647 to 2,146,482,647 ( $-2^{31}-1$  to  $2^{31}-1$ ). A variable is declared a LongInt with:

```
var
  L : LongInt;
```

LongInt and Integer are fully compatible as long as you don't try to assign to an Integer a value out of the Integer range. Doing so will cause a run time error. MacPascal converts all Integer val-

ues to a LongInt for all arithmetic operations. If the value is assigned to an Integer it is then converted back. This need not concern a programmer unless the result of an expression returns a value too large or small to be assigned to an Integer variable. LongInt is an ordinal type and all functions that can be used with an Integer can be used with a LongInt also. If you are familiar with UCSD Pascal then you know that this type has no relation to that Long Integer type.

## The Extended Real Types

Just like Integers, the range of values that can be assigned to a Real is limited. Reals are represented in a mantissa (fractional part) and exponent format. The range of positive values that can be stored in a Real variable is  $1.5 \cdot 10^{-45}$  to  $3.4 \cdot 10^{38}$ . This is a wide range but only provides accuracy to 7 or 8 decimal digits. For greater accuracy, the Double and Extended types can be used.

The range of all Real types are:

| Type     | Range   | Accuracy In Decimal Digits |
|----------|---|----------------------------|
| Real     | $1.5 \cdot 10^{-45}$ to $3.4 \cdot 10^{38}$     | 7-8                        |
| Double   | $5.0 \cdot 10^{-324}$ to $1.7 \cdot 10^{308}$   | 15-16                      |
| Extended | $1.9 \cdot 10^{-4951}$ to $1.1 \cdot 10^{4932}$ | 19-20                      |

Any negative number whose absolute value falls within these ranges can also be represented.

The use of these real types is analogous to what happens with Integer and LongInt types. All of the three real types are fully compatible, but you can't assign a value to a variable if that variable falls outside the range of the variable's type. Before any real operations are performed all values are converted to Extended. The answer is then converted to whatever real type is needed. The following example may clarify this situation.

```
var
  R : Real;
  D : Double;
  E : Extended;
.
.
R := D + E + R;
```

In this example, the values of R and D will be converted to Extended values before addition and an Extended result is produced. The result will then be converted to a Real to be assigned to the Real variable R. This will cause no problem unless the value of the result falls outside the range of values that can be represented by a Real. In general, you should use the type Real unless you need the super accuracy of the Double or Extended types, since the storage and speed requirements of these types are substantial. This system provides the best of both worlds. The high accuracy needed for scientific and statistical work is available without imposing any burden on the programmer.

A fourth Real type, called Computational, is provided for applications such as accounting, that require calculations to be done without any rounding errors being introduced into the fractional part of the number. With the Computational type, values are stored and calculations are done as decimal numbers without any decimal points. No rounding errors will occur as long as the values are in the range of  $-2^{63}-1$  to  $2^{63}-1$ . The largest possible value being 9,223,372,036,854,775,807. This number is larger than the American National Debt multiplied by 1 million. Since values are stored without a decimal point, dollars and cents can be represented by assuming a decimal point between the second and third digits. Computational values can be displayed with a decimal point placed between any two digits. The second fieldwidth parameter is used to specify what digit to place the decimal point to the left of. For instance,

```
var
```

```
  C : Computational;
```

```
  C := 12345; {assumed to be 123.45}
```

```
  Writeln(C : 5 : 2);
```

will display

```
  123.45
```

**Note:** In Release 1.0 of MacPascal this feature did not work properly. In future releases it will work as described.

## The Arithmetic Functions

The SQR function returns the square of a number. The value used can either be a Real or an Integer. The result is either a



LongInt or an Extended Real:

SQR(5)        returns 25  
SQR(1.3)      returns 1.69

The SQR function returns the square root of the value given. The value (argument) can be either a Real or an Integer, the result is always an Extended Real.

SQRT(9)       returns 3.0  
SQRT(30)      returns 5.5e+0

The ABS function returns the absolute value of the value given. The absolute value of a number is that number regardless of its sign. The argument may be either a Real or an Integer, the result is always the same type as the argument.

ABS(5)        returns 5  
ABS(-5)       returns 5  
ABS(0)        returns 0  
ABS(-1.32)    returns 1.32

The ODD function tests an Integer to see if it is odd or even. A Boolean is returned. TRUE, if the Integer is odd, FALSE, if the Integer is even.

ODD(1)        returns TRUE  
ODD(6)        returns FALSE

## The Trigonometric Functions

The SIN, COS and ARCTAN functions are used for trigonometric operations. They take either a Real or an Integer as an argument and assume it to be an angle expressed in radians. The result is always an Extended Real.

COS(x)        returns the Cosine of x.  
SIN(x)        returns the Sine of x.  
ARCTAN(x)     returns the Arctangent of x.

Any of the other trigonometric functions can be synthesized by using the existing ones.

To find the Tangent of x,       $\text{Sin}(x)/\text{Cos}(x)$   
To find the Cosecant of x,      $1/\text{Sin}(x)$

## The Logarithmic Functions

Pascal has the EXP and LN natural logarithmic functions. Both take either an Integer or a Real as an argument and always return an Extended Real.

|        |                                     |
|--------|-------------------------------------|
| LN(x)  | returns the natural logarithm of x. |
| EXP(x) | returns the value of $e^x$ .        |

## Tool Box Functions

The Macintosh Toolbox contains several functions that are not part of standard Pascal. Three functions SysBeep, Page, and TickCount are presented here. Other Toolbox features will be introduced in other chapters of the book.

The SysBeep function causes a tone to be generated on the Macintosh's speaker. The duration of the tone is determined by the Integer argument used times .022 seconds.

**SysBeep(100);** creates a tone that will last 2.2 seconds.

The Page function causes the Text Window to be cleared.

**Page;** clears the Text Window.

The TickCount function returns a Long Integer representing the elapsed time since the machine was turned on in 1/60ths of a second. This information is probably of little use by itself, but Tickcount can be used to time the execution of MacPascal statements. For instance, we could measure the execution time of a for loop with:

```
Start := TickCount;
for I := 1 to 10 do
  Sum := Sum + 1;
Stop := TickCount;
ElapsedTime := Stop - Start;
```

The first call to TickCount marks the elapsed time before the loop starts. The second call marks the elapsed time after the loop terminates. The difference between the two represents the amount of time needed to execute the loop. For the curious, the execution time is 2/60<sup>th</sup>s of a second.

## User-Defined Data Types

When programming in Pascal, the programmer is not limited to using the standard Pascal data types. The programmer is free to declare his own ordinal data types with the **type** statement. Any programmer declared data type of this kind is called a User-defined or Enumerated type. The **type** statement appears between the **const** and the **var** statements.

```

program Typexample;
  const
    C=1;
  type
    Days = (Sun, Mon, Tue, Wed, Thur, Fri, Sat);

```

This declares a new data type called Days. The set of possible values of a variable of type Days are enumerated in the parentheses in the statement.

The syntax diagram of the **type** statement is:

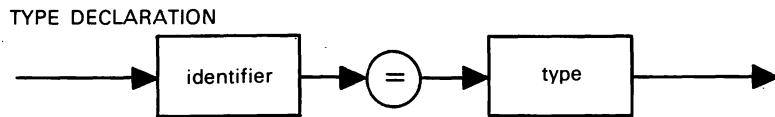


Figure 6-1. Syntax Diagram : Type Declaration

Variables can now be created of type Days.

```

var
  PayDay: Days;

```

The variable PayDay is of data type Days. The operations that can be performed with a User-defined type are limited but several are possible. An assignment statement looks like you might expect it to.

```

  PayDay := Thur;

```

Notice that there are no quotation marks around Thur. This is because it is not a string value and should not be confused with one. It is one of the possible values that can be assigned to a variable of the data type Days. This is not any different from using an assignment statement with a Boolean variable.

```

  Switch := TRUE;

```



Here, one of the possible Boolean values is assigned to the Boolean variable Switch. As a matter of fact, the type Boolean can be thought of as a pre-declared user-defined type with the declaration.

**type**

```
Boolean = (FALSE, TRUE);
```

The type Boolean is declared with its two possible values.

Since user-defined types are ordinal types, the SUCC and PRED functions are available. Assuming the PayDay := Thur;

|              |  |
|--------------|--|
| SUCC(PayDay) | returns Fri                              |
| PRED(PayDay) | returns Wed                              |
| PRED(Sun)    | is undefined and causes a run time error |
| SUCC(Sun)    | returns Mon                              |

When given a User-defined type the ORD function returns an Integer representing the value's position in the declared list of values. The ORD of the first value declared is zero. Here is the declaration of type Days and the ORD of the values.

```
Days = ( Sun, Mon, Tue, Wed, Thur, Fri, Sat);
ORDs   0   1   2   3   4   5   6
```

The following loop will print the ORDs.

```
for PayDay:= Sun to Sat do
  Writeln( Payday, ORD(PayDay));
```

Displayed by the loop is:

|      |   |
|------|---|
| Sun  | 0 |
| Mon  | 1 |
| Tue  | 2 |
| Wed  | 3 |
| Thur | 4 |
| Fri  | 5 |
| Sat  | 6 |

The value of a variable with a user-defined type can also be printed. This is a special feature of MacPascal and is not present in most Pascal implementations. The following loop will display the values of the Days data type.

```
for PayDay:= Sun to Sat do
  Writeln(PayDay);
```

Displayed is:

Sun  
Mon  
Tue  
Wed  
Thur  
Fri  
Sat

When two user-defined values are compared their Ords are used to determine the greater. For instance, Tue is greater than Sun because the ORD(Tue) is greater than ORD(Sun).

User-defined types are a feature not available in any other popular programming language. While they might appear trivial, they are a powerful programming tool that will make a program easier to write, read, and debug. They should be used where ever possible in your programming.

A short cut exists to define a User-defined type. The possible values are listed directly next to the variable eliminating the type statement. This declaration:

```
program TypeExample;  
  type  
    Days = (Sun, Mon, Tue, Wed, Thur, Fri, Sat);  
  var  
    PayDay: Days;
```

can be replaced by:

```
program Typexample;  
  var  
    PayDay: (Sun, Mon, Tue, Wed, Thur, Fri, Sat);
```

This method should not normally be used because it affects the program's readability.

## Subranges

A subrange is a subset of a declared ordinal data type. Here are some examples:

```
type  
  Letters = 'A'..'Z';  
  Cards = 1..52
```

**var**

Deck : Cards;

Ch : Letters;

Two subranges, Letters and Cards, have been declared as subsets of their types. Letters of type CHAR and Cards of type INTEGER. The first and last values of the range is given with the two periods (..) standing for all the values in between. The syntax diagram for a subrange is:

SUBRANGE TYPE

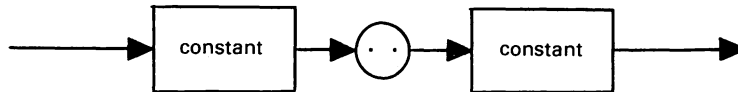


Figure 6-2. Syntax Diagram : Subrange Type.

Any attempt to assign a value other than one included in the subrange to any of the variables would result in an error. The philosophy behind this is that it is preferable to have an error occur and stop a program than to let the program continue with invalid data. This is an example of Pascal's strong typing, the careful checking of value and data types for compatibility. It is a good programming practice to use subranges where ever a set of possible values will fall into a predictable range. Examples of this might be exam grades, temperatures, ages, and countless others.

Subranges can also be declared for user-defined types.

**type**

Days = (Sun, Mon, Tue, Wed, Thu, Fri, Sat);

Weekdays = Mon..Fri;

Here the type Weekdays is a subrange of the user-defined type Days.

The **type** statement can also be used to redefine an existing data type with a new name.

**type**

Numbers= Integer;

**var**

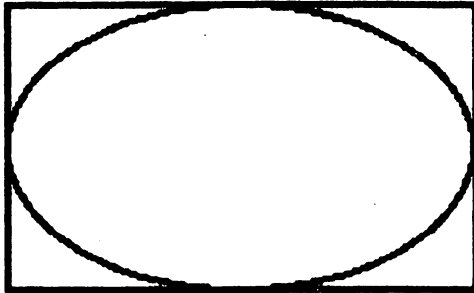
I, J, K : Numbers;

In this example, the variables I, J, and K are all of the type Numbers which are equivalent to the type Integer, they just have a different name. Some programmers do this to improve the readability of their programs, but this is not a common technique.



## Drawing Ovals

Along with drawing rectangles MacPascal has two ways of creating ovals with Quickdraw. The first method utilizes the oval commands, `FrameOval`, `EraseOval` and `PaintOval`. These are analogous to the `FrameRect`, `EraseRect` and `PaintRect` commands. An oval is defined as the largest ellipse that can be inscribed inside a particular rectangle.



**Oval inscribed  
in a rectangle**

Figure 6-3.

To draw a circle, define a rectangle that is a square (all sides equal) and then use `FrameOval`.

```
SetRect(R,10,10,60,60);  
FrameOval(R);
```

The display in the **Drawing** window is shown in Figure 6-4.

The radius of the circle is one half the length of a side of the square.

To display an ellipse, define a non-square rectangle instead (see Figure 6-5).

```
SetRect(R,10,10,60,80);  
FrameOval(R);
```

The `PaintOval` and `EraseOval` commands operate the same way.

A second method of drawing a circle is to use the `PaintCircle` command which displays a circle of a given radius centered at a given point, filled with black.

The command:

```
PaintCircle(10,20,5);
```

displays a circle of radius 5 centered at point 10,20.

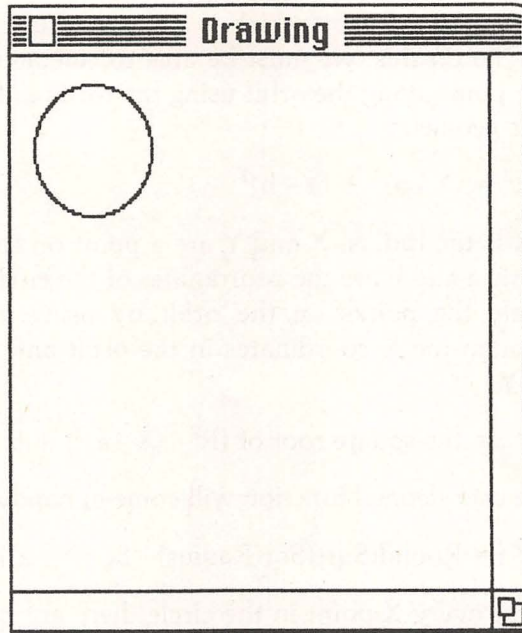


Figure 6-4.

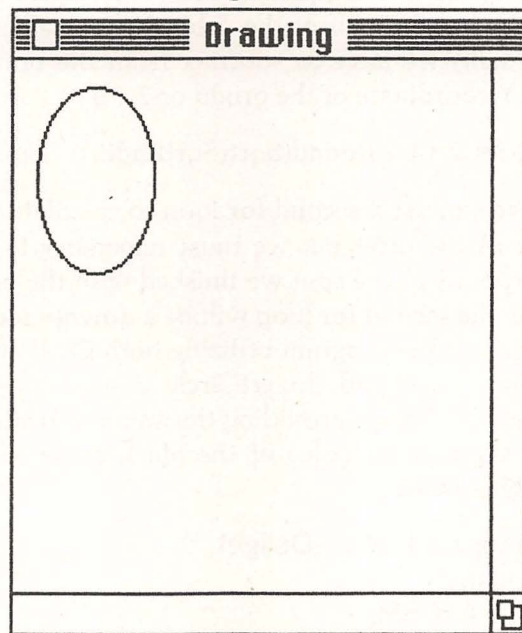


Figure 6-5.

A demonstration that comes to mind when drawing ovals is that of a moon rotating around a fixed planet. The moon will be

oriented in an orbit which is a fixed radius from the planet's center. To do this, we must be able to calculate the coordinates for any point along the orbit using the formula for a circle from analytic geometry.

$$R^2 = (X - a)^2 + (Y - b)^2$$

R is the radius, X and Y are a point on the circle and the constants a and b are the coordinates of the circle origin. We can calculate the points on the orbit by using a **for** loop to iterate through the X coordinates in the orbit and solving the equation for Y.

$$Y := \text{the square root of } (R^2 - (X - a)^2) + b$$

The user-defined function will come in handy in calculating Y:

$$Y := \text{Round}(\text{Sqrt}(\text{Sqr}(\text{Radius}) - \text{Sqr}(X - a)) + b);$$

For every X point in the circle there are two Y points. One on the bottom half of the circle and the other at the same position on the top half. The equation above will calculate the Y coordinates of the bottom half of the orbit, but to calculate the points in the top half, we have to subtract from the bottom half point twice the Y coordinate of the origin or  $2 * b$ .

$$Y := 2 * b - \text{Round}(\text{Sqrt}(\text{Sqr}(\text{Radius}) - \text{Sqr}(X - a)) + b);$$

We can use a second **for** loop to calculate the points on the top half of the orbit but we must remember to start calculating the top points at the spot we finished with the bottom points. Therefore, the second **for** loop will be a **downto** loop.

Here is the program utilizing both **Oval** and **Circle** commands. A new command, **InvertCircle** is used to erase the moon right after it is drawn, providing the animation effect. **InvertCircle** simply changes the color of the black circle to white, which essentially erases it.

```

program KeplersDelight;
  const
    A = 55;
    B = 55;
  var
    R : Rect;
    Radius, X, Y : Integer;

```



```

begin
  Radius := 45;
  SetRect(R, 45, 45, 65, 65);
  FrameOval(R); {Draw the planet}
  {Calculate Y points in bottom half of orbit}
  for X := 10 to 100 do
    begin
      Y := Round(Sqrt(Sqr(Radius) - Sqr(X - A)) + B);
      PaintCircle(X, Y, 5);
      InvertCircle(X, Y, 5)
    end;
    {Calculate Y points in top half of orbit}
  for X := 100 downto 10 do
    begin
      Y := 2 * B - Round(Sqrt(Sqr(Radius) - Sqr(X - A)) + B);
      PaintCircle(X, Y, 5);
      InvertCircle(X, Y, 5)
    end
  end.

```

Run the program and watch the moon orbit the planet. It may be helpful to set the **Observe** window with the variables X and Y and watch their values as they are calculated. You might want to try to adapt the program to use either all Oval or Circle commands. A real challenge to those comfortable with analytic geometry would be to have a smaller moon orbit the larger moon as that moon orbits the planet.

## Exercises

1. Determine whether each of these expressions produces a Real or Integer value.
  - a.  $10 / 3 + 5 * 2$
  - b.  $\text{Trunc}(3.47) + 4.0$
  - c.  $\text{Round}(10 \text{ div } 3) + 4$
  - d.  $\text{ORD}('A') + 4$
2. Evaluate the following expressions to find the answer.
  - a.  $15 + \text{SQR}(6)$
  - b.  $\text{Round}(3.4) + 17$
  - c.  $\text{ABS}(-13) + \text{SQRT}(16)$
  - d.  $\text{ORD}(\text{TRUE}) + 2$

3. Write the following expressions in Pascal.
  - a.  $X^2 + Y$
  - b.  $\frac{AX + B}{D^2}$
  - c.  $\text{LN}(15 + e^X)$
  - d.  $\text{TAN}(3\pi)$
4. Define an enumerated type for each of the following:
  - a. The positions on a baseball team
  - b. Your family members
  - c. Types of pizzas

# 7 Procedures

Pascal provides a way to break a large program into smaller sections for easier design and programming. These sections are called procedures. Procedures have a structure similar to programs and are therefore often referred to as subprograms.

Actually, procedures are one of the two types of subprograms used in Pascal, the other being user-defined functions. Subprograms are included inside a program, but their instructions remain separate from the instructions of the main program.

```
program example;  
  var  
    X, Y, Z : Real;  
  procedure Add;  
  begin {Procedure}  
    Z := X + Y;  
  end; {Procedure}  
begin {Main Program}  
  Writeln('Enter two numbers');  
  Readln(X, Y);  
  Add;  
  Writeln('The sum is', Z : 6 : 2)  
end. {Main Program}
```

This program contains a procedure named Add. The name of a procedure is far more important than the name of a program, though the naming of both follows the same syntactical rule. The program name is similar to a comment not serving any real purpose in the program. The procedure name has a purpose, it is used to identify and differentiate procedures in a program.

A procedure is declared by placing the **procedure** statement right after the **var** section of a program. The structure of a proce-



cedure is analogous to that of a program except that it starts with the word **procedure** rather than **program** and the final **end** in a procedure is followed by a semicolon rather than a period.

A procedure can contain any statement that can be used in a program. One exception is that the final **end** in a procedure is followed by a semicolon rather than a period. As always, execution of the program starts with the first statement in the main program. While the main program executes, the procedure lies dormant waiting to be called. In the example, the first statements executed are the `Writeln` and the `Readln`. The third line in the main program doesn't contain a statement but rather has the name of the procedure, `Add`. This is how a procedure is called, by using the procedure's name as a statement. A procedure must be defined before it is called.

When a procedure is called the main program goes into a dormant, waiting state and the procedure is awoken. The statements in the procedure now start to execute beginning with the first statement. This example has only one statement in the procedure that adds together the two values that were read by the main program. When the end of the procedure is reached (the last **end** statement in the procedure), control is passed back to the main program which resumes execution at the line after the one that called the procedure.

This was a short and simple example of procedures, but yet it illustrated how procedures are written and called. Programmers use procedures to help clarify the design and writing of a program. Usually the first step in programming is to describe the steps needed to solve the problem in an English-like form known as pseudocode. Here is the pseudocode for a program that calculates the area and perimeter of a rectangle.

```
Read the length and width  
Find the perimeter  
Find the area  
Print the answer
```

Notice that the pseudocode doesn't include the actual Pascal statements needed, but just the steps involved. Depending upon the complexity of the program, the pseudocode may then be written several times in increasing detail, but in this simple case we can begin to build the Pascal program.

```

program Rectangle;
  var
    Len, Width, A, P : Real;
  {Procedures go here}
begin
  Writeln('Enter the length and width');
  Readln(Len, Width);
  Perimeter;
  Area;
  Writeln('The area is', A : 6 : 2, 'The perimeter is', P : 6 : 2)
end.

```

This second step in the design process results in a part of a Pascal program that just shows the variables and the main section of the program. Notice the calls to the two procedures named *Area* and *Perimeter*, which have yet to be written. The next step is to write the procedures and include them in the program.

```

program Rectangle;
  var
    Len, Width, A, P : Real;
  procedure Area;
  begin {Area}
    A := Len * Width;
  end; {Area}
  procedure Perimeter;
  begin {Perimeter}
    P := 2 * (Len + Width)
  end; {Perimeter}
begin
  Writeln('Enter the length and width');
  Readln(Len, Width);
  Perimeter;
  Area;
  Writeln('The area is', A : 6 : 2, 'The perimeter is', P : 6 : 2)
end.

```

This method of programming, sometimes called structured programming or top down programming, emphasizes planning a program prior to writing it and creating a main program consisting mostly of procedure calls. This style of program planning has replaced the use of flowcharts when programming in structured languages such as Pascal.

## Scope of Variables

In the procedure demonstrated before, the variables used were declared in the main program. However, as was mentioned before, a procedure can have any statements that are allowed in a main program. This means that a procedure can also have its own variable declarations. This presents a puzzling question. When a procedure has its own variable declaration where and when can its variables be used and how about the variables declared in the main program?

It was obvious from the example programs that a procedure may use the variables declared in the main program, but can the main program use the variables declared in a procedure? The answer to this is no. A procedure's variables are active (can be used) if they are declared in that procedure or in any other variable declaration in the part of the program containing the procedure. Consider Figures 7-1 and 7-2 which describe the scope of variables in two programs.

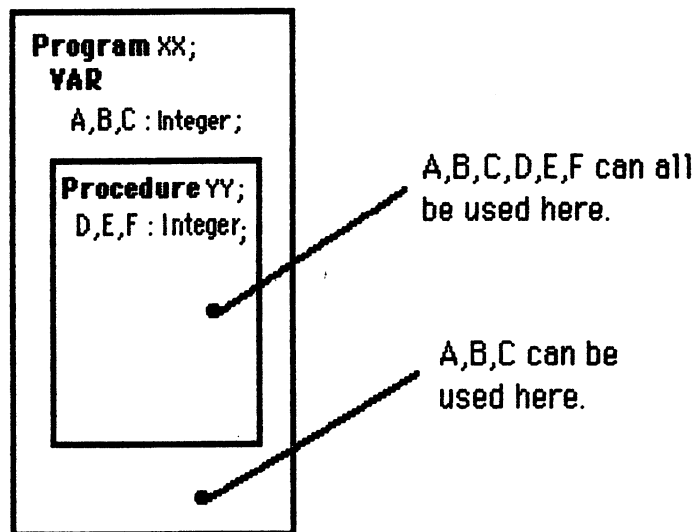


Figure 7-1.

When a procedure that declares variables becomes active (is called), the variables declared are created and given their initial value (depending on the data type). While that procedure executes, the variables are live, but once the procedure ends, the variables are destroyed and can't be accessed. Even when a proce-



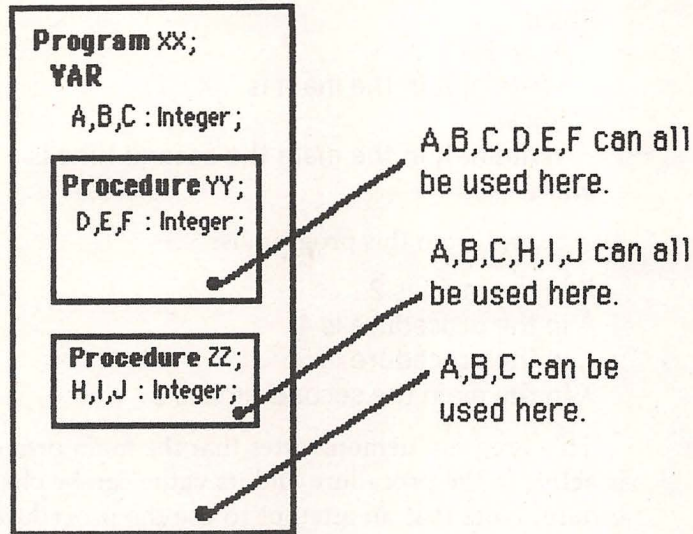


Figure 7-2.

cedure is called multiple times, the variables are created and destroyed each time it is called. This means that a value can't be left in a procedure for subsequent calls to that procedure. Since the main program is always active (until the final **end** is encountered) its variables are always available for use.

A variable that is declared in a procedure is known as a local variable, since it is active only in that procedure. Variables declared in the main program are known as global variables, designating that they are active throughout the entire program.

Let's examine the following program to understand the difference between local and global variables.

```

program Proceed;
var
  X : Integer;
procedure Show;
var
  A : Integer;
begin {Procedure}
  A := 4;
  Writeln( 'A in the procedure is ', A : 1);
  X := X + 3;
  Writeln('X in procedure is ', X : 1)
end; {Procedure}
  
```

```
begin
  X := 2;
  Writeln('X in the main is ', X : 1);
  Show;
  Writeln('X in the main the second time is ', X : 1)
end.
```

The output from this program is:

```
X in the main is 2
A in the procedure is 4
X in the procedure is 5
X in the main the second time is 5
```

This program demonstrates that the main program's variable X is active in the procedure and its value can be changed in the procedure. Note that an attempt to use the procedure's variable A in the main program such as:

```
Writeln(A)
```

would produce an error since the procedure's variables are not active in the main program. Let's change the program and call the procedure more than once.

```
program Proceed;
var
  X : Integer;
procedure Show;
var
  A : Integer;
begin {Procedure}
  A := A + 3;
  Writeln( 'A in the procedure is ', A : 2)
end; {Procedure}
begin
  Show;
  Show
end.
```

The output is:

```
A in the procedure is 3
A in the procedure is 3
```

This program took advantage of the fact that MacPascal creates all local variables when a procedure is called. Note that even

though the procedure is called twice, the printed values of A are the same. If this was not the case, the second value of A would be 6 not 3.

Local variables inside a procedure may even have the same name as a global variable. For example:

```

program TwoNames;
  var
    X : Integer;
  procedure TheSame;
    var
      X : Integer;
    begin
      X := 1;
      Writeln(X)
    end; {Procedure}
  begin
    X := 4;
    Write(X);
    TheSame;
    Writeln(X)
  end.

```

The output is:

```

4
1
4

```

If a Local and Global variable have the same name, the local variable is the one that is used. Hence, in the procedure the local variable named X was used and the value of the Global variable named X was not changed. This exercise was used only to emphasize the point and constitutes a bad programming practice due to the confusion created.

One last note about local variables. When using a **for** loop inside a procedure the control variable of the **for** loop must be a local variable.

## Parameter Passing

In the programs we have seen, main programs have provided procedures with information via Global variables and procedures



have sent information back to the main program the same way. There are, however, limits to this system that can be demonstrated with the following program.

```
program Demo;  
  var  
    A, B, C, D : Integer;  
  procedure Swap;  
    var  
      Temp : Integer;  
  begin  
    Temp := A;  
    A := B;  
    B := Temp  
  end; {Procedure}  
begin {Main}  
  A := 4;  
  B := 3;  
  C := 5;  
  D := 1;  
  Swap;  
  Writeln(A : 2, B : 2)  
end.
```

The output of this program is: 3 4

In this program, procedure Swap is used to exchange the values of variables A and B. After Swap is called, A has a value of 3 and B has a value of 4. Notice that the information was sent back and forth between the main program and the procedure with the use of Global variables ( A and B). The problem this presents is that this procedure can not now be used to exchange the values of C and D or any other set of variables. We can say that the variables A and B are hard coded into the procedure. In order to swap any two other variables a new procedure would need to be written that had those variables hard coded in it. The inefficiency of this situation can be solved with the use of parameters. Parameters allow variables in the main program to be substituted for the actual variables specified in the procedure. This allows the same procedure to be used with different sets of variables. There are two types of parameters in Pascal, variable and value. First let's examine variable parameters.

## Variable Parameters

The parameters to be used in a procedure are listed as part of the procedure statement.

```
procedure Swap(var E, F : Integer);
```

Here two parameters E and F are designated. They are both of the type Integer. Placing variable names in the parameter list is similar to declaring them as local variables in that they are active as long as the procedure is active and they have the same scope (where they are active). The difference is that when a variable parameter is created, it is given an initial value from the variables listed in the statement that calls the procedure.

When using parameters, the way a procedure is called changes slightly.

```
Swap(A, B);
```

This call to the procedure Swap now includes a list of variables which is to be matched up with the list of parameters in the procedure heading. Notice that the variables listed in the call to the procedure are not the same as the ones listed in the procedure statement, but rather are variables in the main program whose values we want to send to the procedure. A relationship exists between these two lists of variables (Figure 7-3). The variables listed in the procedure heading are called Formal parameters, the variables listed in the call to the procedure are called Actual parameters. The Actual and Formal parameters must agree in number and type and their position in the list is important. When the procedure is called, the Formal and Actual parameters are matched the first Formal parameter with the first Actual parameter, the second Formal parameter with the second Actual parameter, and so on. When using variable parameters the variable declared as a Formal parameter is not created as a memory location but rather becomes a pointer to its corresponding Actual parameter. This essentially creates two names for the same memory location.

We say that E and F point to the variables A and B. When in the procedure a change is made to E, it is actually made to the variable that E points to which is A. When a change is made to F it is actually B that is affected. When a change in a Formal parameter is reflected in an Actual parameter we call it a side affect. Here is the entire program rewritten to include parameter passing. Several Writeln statements have been added to help clarify the situation.

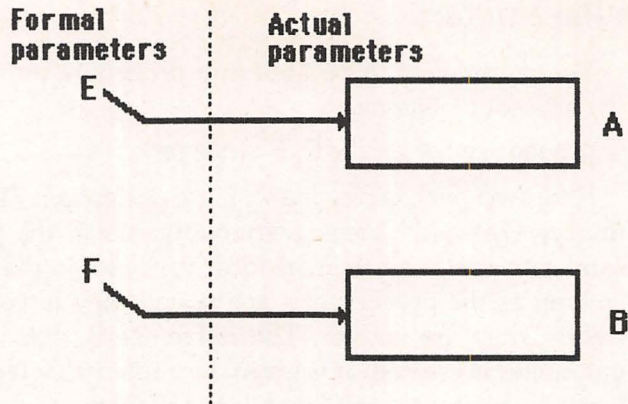


Figure 7-3.

```

program Demo;
  var
    A, B, C, D : Integer;
  procedure Swap(var E, F : Integer);
    var
      Temp : Integer;
  begin
    Writeln('E =', E : 1, ' F =', F : 1);
    Temp := E;
    E := F;
    F := Temp;
    Writeln(' E =', E : 1, ' F =', F : 1)
  end; {Procedure}
begin {Main}
  A := 4;
  B := 3;
  C := 5;
  D := 1;
  Writeln('A =', A : 1, ' B =', B : 1);
  Swap(A, B);
  Writeln('A =', A : 1, ' B =', B : 1)
end.

```

The output is

```

A = 4   B = 3
E = 4   F = 3
E = 3   F = 4
A = 3   B = 4

```



From the output you can see where the Formal parameters were exchanged and how it affected the Actual parameters in the main program.

## The Major Advantage

The major advantage of passing information to a procedure with parameters rather than with Global variables is that the same procedure can be used with different sets of variables. Procedure Show can be used to exchange the values of C and D by just changing the Actual parameters used in the procedure call.

```
Swap(C, D);
```

## Value Parameters

A second mechanism for passing parameters is the use of value parameters. When value parameters are used there are no side effects in the main program. With value parameters, a copy of the Actual parameters are created and assigned to the Formal parameters. Hence any changes made to the copies of the original memory locations do not affect the original locations themselves. Value parameters are also declared in the procedure heading except the word `var` is eliminated.

```
procedure Swap(E, F : Integer);
```

Now E and F are value parameters rather than variable parameters. When the procedure is called:

```
Swap(A, B)
```

the corresponding Actual parameters are copied into the Formal parameters (Figure 7-4).

Now, when the values of the Formal parameters are changed inside the procedure, the changes are not reflected in the main program's variables which are Actual parameters. Therefore, value parameters can not be used to send information from a subroutine to a main program. Global variables and variable parameters must be used for this purpose.

Because only a value is sent to the procedure when using value parameters, the value used need not be contained in a variable as it must be in a variable parameter. A constant or expression can be used in the Actual parameter list. Some examples of this are:

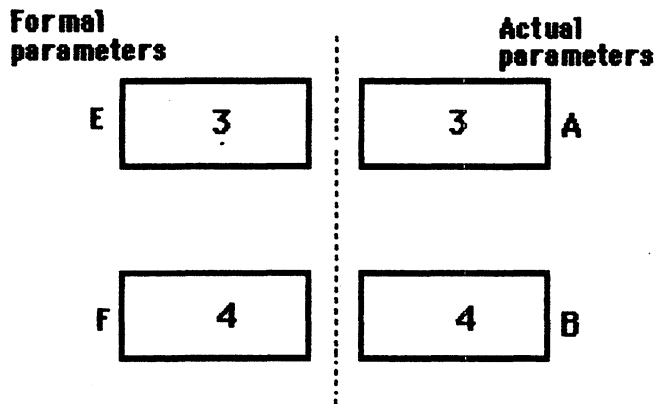


Figure 7-4.

```
Proc1(A, 2, A * 3)
```

Let's now see some more examples of the use of value parameters.

```

program ValExamples;
  var
    K, L, M : Integer;
  procedure Lines(N : Integer);
    var
      I : Integer;
    begin
      for I := 1 to N do
        WriteLn
      end; {Lines}
  procedure Spaces(N : Integer);
    var
      I : Integer;
    begin
      for I := 1 to N do
        Write(' ':1)
      end; {Spaces}
  begin {main}
    K := 5;
    Write(K : 1);
    Spaces(3);
    Write(K + 1 : 1);
    Lines(2);
    Write(K : 1)
  end.
```

The output would look like:

```
5   4
4
```

The procedures `Spaces` and `Lines` do not send any values back to the main program and as such make use of value rather than variable parameters. These procedures are used to format output in the `Text` window. Normally, anytime you know that the Actual parameter should not be allowed to be changed by a procedure, you should use value parameters.

## Comparing Value and Variable Parameter Passing

To emphasize the difference between the two ways of parameter passing, here are two versions of the same program, one using variable parameters, the other value parameters.

|   |  |
|---|--|
| <pre> <b>program</b> Vars;   <b>var</b>     X, Y : Integer;   <b>procedure</b> PT(<b>var</b> A, B : Integer);   <b>begin</b>     A := A + 1;     B := B + 2;     Writeln(A : 2, B : 2);   <b>end</b>; <b>begin</b> {main}   X := 1;   Y := 1;   Writeln(X : 2, Y : 2);   PT(X, Y);   Writeln(X : 2, Y : 2) <b>end</b>. </pre> | <pre> <b>program</b> Vals;   <b>var</b>     X, Y : Integer;   <b>procedure</b> PT(X, Y : Integer);   <b>begin</b>     A := A + 1;     B := B + 2;     Writeln(A : 2, B : 2);   <b>end</b>; <b>begin</b> {main}   X := 1;   Y := 1;   Writeln(X : 2, Y : 2);   PT(X, Y);   Writeln(X : 2, Y : 2) <b>end</b>. </pre> |
|---|--|

The outputs are:

|  |   |
|--|---|
| <pre> 1  1 2  3 ← from procedure 2  3 </pre> | <pre> 1  1 2  3 ← from procedure 1  1 ← no side effect </pre> |
|--|---|

As you can see in the program on the left, variable parameters were used and the variables in the main program were affected by



the operations in the procedure. In the program on the right, value parameters were used and there was no effect on the variables in the main program.

## Mixing Variable and Value Parameters

In many situations you will mix the use of variable and value parameters. Both types can be included in the procedure's parameter list. Each separate declaration of variable parameters must have its own `var` in front of it.

`procedure Mixed (var X : Integer; Y, Z : Real; var P : Boolean);`

The following example raises a number to a power (a mathematical operation left out of Pascal). It is a typical example of mixing both value and a variable parameter in a procedure. The information sent to the procedure, the number, and the power, are both value parameters. The answer is sent back to the main program in a variable parameter (`Ans`).

```

program PowerTest;
  var
    Num, Power : Integer;
    Ans : Real;
  procedure RaiseToPower(var Ans : Real; Base, Power :
                        Integer);
    var
      I : Integer;
  begin
    Ans := 1;
    for I := 1 to Power do
      Ans := Ans * Base
  end;
begin
  Writeln('Enter the number and the power to raise it to');
  Readln(Num, Power);
  RaiseToPower(Ans, Num, Power);
  Writeln(Ans)
end.

```

This procedure worked by multiplying the Base by itself Power number of times. Lets trace the procedure for 2 and 3 as the values of Base and Power.

| I | Ans | Base | Power |        |
|---|-----|------|-------|--------|
| — | 1   | 2    | 3     |        |
| 1 | 2   | 2    | 3     |        |
| 2 | 4   | 2    | 3     |        |
| 3 | 8   | 2    | 3     | ← Done |

## Mortgage Calculator

Let's now turn to a large, more complex application that requires the use of procedures to help organize the program development process. From time to time people find themselves in a situation where they have to borrow a substantial sum of money. Loans of this type, of which a mortgage is one (the Latin translation of mortgage is death commitment, if you have a house you know what this means), are known as amortized loans. In an amortized loan the monthly payment is constant throughout the life of the loan, but the part of the payment that goes towards interest and the part that goes towards reducing the amount borrowed (Principal) varies. In the early years of the loan, the interest component of the payment is high meaning very little Principal is paid off. Little by little the Principal is reduced which decreases the interest component and increases the principle part of the payment. This turns out to be rather complicated by analysis and so it would be helpful to have a program to calculate the payments. The program can not only calculate the monthly payment and the yearly Principal and interest paid, but can also report the true cost of the payments after considering the income tax deduction on the interest paid and subtracting that from the total yearly payment.

The formulas for the calculations are:

$$\text{Annual payment} = \frac{\text{Interest rate} * (\text{Interest rate} + 1)^{\# \text{years}} * \text{Principal}}{(\text{Interest rate} + 1)^{\# \text{of years}} - 1}$$

$$\text{Monthly Payment} = \text{Annual payment} / 12$$

$\text{Yearly Interest paid} = \text{Interest rate} * \text{Principal remaining}$

$\text{Yearly Principal paid} = \text{Annual Payment} - \text{Yearly interest paid}$

$\text{Principal Remaining} = \text{Principal} - \text{Yearly Principal paid}$

$\text{Actual Cost} = \text{Interest paid} * (1 - \text{Tax bracket}/100 + \text{Principal paid})$

The program will prompt the user to enter

The Principal amount

The Interest rate (as a fraction)

The Term of loan in years

The Tax Bracket (as a percentage)

First let's pseudocode the program.

Get input

Calculate annual and monthly payment

For I:= 1 to Years Do

begin

calculate yearly Principal and interest paid

calculate yearly cost

write information

end.

Now let's take a stab at writing the main program without the procedures.

**program** Mortgage;

**var**

Years, I : Integer;

AnPay, MonthPay, Principal, AnPrinPay, AnCost,

AnInterestPay, IntRate, TaxBrac : Real;

{Procedures here}

**begin**

GetInfo;

CalculatePayment(AnPay, MonthPay, Years, IntRate);

Writeln('The monthly payment is ',MonthPay : 8 : 2);

PrintTable

**end.**

Now let's put everything together.



```

program Mortgage;
var
    Years, I : Integer;
    AnPay, MonthPay, Principal, AnPrinPay, AnCost,
    AnInterestPay, IntRate, TaxBrac : Real;
procedure GetInfo;
begin
    Writeln('Enter amount of loan');
    Readln(Principal);
    Writeln('Enter number of years');
    Readln(Years);
    Writeln('Enter interest rate as a fraction');
    Readln(IntRate);
    Writeln('Enter your tax bracket');
    Readln(TaxBrac)
end;
{-----}
procedure CalculatePayment (var AnPay, MonthPay :
                                Real;
                                Years : Integer;
                                IntRate : Real);
var
    TempResult : Real;
    I : Integer;
begin
    TempResult := 1;
    for I := 1 to Years do
        TempResult := TempResult * (IntRate + 1);
    AnPay := (IntRate * TempResult * Principal) /
        (TempResult - 1);
    MonthPay := AnPay / 12
end;
{-----}
procedure PrintTable;
var
    Temp, I : Integer;
begin
    Writeln('Year', ' ' : 1, 'Interest', ' ' : 3, 'Princ', ' ' : 5,
        'Cost');
    for I := 1 to Years do
        begin
            Write(I : 3);

```

```

        AnInterestPay := Principal * IntRate;
        Write(AnInterestPay : 10 : 2);
        AnPrinPay := AnPay - AnInterestPay;
        Write(AnPrinPay : 10 : 2);
        Principal := Principal - AnPrinPay;
        AnCost := (TaxBrac / 100 * AnInterestPay) +
            AnPrinPay;
        Writeln(AnCost : 10 : 2)
    end {For Loop}
end; {Procedure}
{-----}
begin
    GetInfo;
    CalculatePayment(AnPay, MonthPay, Years, IntRate);
    Writeln('The monthly payment is ', MonthPay : 8 : 2);
    PrintTable
end.

```

If the values entered were an \$80,000 loan at 13% for a period of 10 years and the borrower's tax bracket was 40% the program would produce this table.

|                        |          |          |          |
|------------------------|----------|----------|----------|
| The monthly payment is | 1228.60  |          |          |
| Year                   | Interest | Princ    | Cost     |
| 1                      | 10400.00 | 4343.17  | 8503.17  |
| 2                      | 9835.39  | 4907.78  | 8841.93  |
| 3                      | 9197.38  | 5545.79  | 9224.74  |
| 4                      | 8476.42  | 6266.74  | 9657.31  |
| 5                      | 7661.75  | 7081.42  | 10146.12 |
| 6                      | 6741.17  | 8002.00  | 10698.47 |
| 7                      | 5700.90  | 9042.26  | 11322.62 |
| 8                      | 4525.41  | 10217.75 | 12027.92 |
| 9                      | 3197.10  | 11546.06 | 12824.90 |
| 10                     | 1696.12  | 13047.05 | 13725.50 |

This program could have been written without using procedures but it would have been harder to organize and write. You will soon find procedures a great aid to you in your future programs.

## Drawing Lines

In previous chapters we have seen how to draw and animate rectangles and ovals in the **Drawing** window. In this section we

will now look at drawing lines. Lines, like the other two graphics structures we have seen are drawn in the **Drawing** window with a QuickDraw tool called the pen. Obviously, there is physically no pen drawing on your screen. The pen is a metaphor for describing drawing operations as though they were done on a paper with an ink pen.

To draw lines, we utilize that old saying about the straight path between two points is a line. First, we position the pen with the **MoveTo** command.

```
MoveTo(X, Y);
```

**MoveTo** picks up the pen and moves it to the point X,Y without drawing anything. Do this to position the pen to the starting point of the line. The line is actually drawn with the **LineTo** command.

```
LineTo(X, Y);
```

**LineTo** draws a line from the old pen position to the new point given in the command.

For example, to draw a line across the window from 10,10 to 10,100 we would:

```
MoveTo(10,10); {move to starting point}  
LineTo(10,100); {draw line}
```

Let's combine drawing lines with procedures to create a procedure that draws an equilateral triangle. An equilateral triangle has three equal sides all connected by 60 degree angles. The procedure will have three variable parameters, X and Y, the upper left hand point in the triangle and Side, the length of the sides. Geometry tells us the coordinates of the other points.

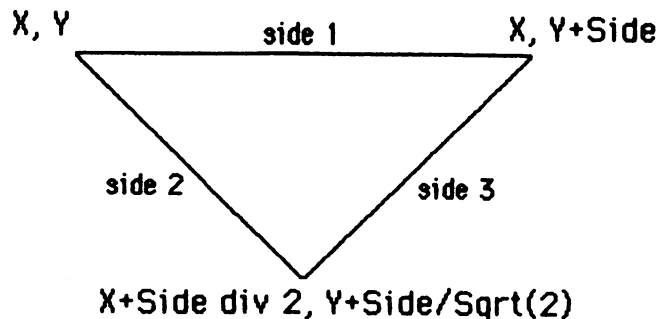


Figure 7-5



In the procedure, we first position the pen at point X, Y. Then draw sides 1, 2 and 3.

```
procedure Tri (X, Y, Side : Integer);  
begin  
  MoveTo(X, Y);  
  LineTo(X + Side, Y);  
  LineTo(X + Side div 2, Round(Y + Side / Sqrt(2)));  
  LineTo(X, Y)  
end;
```

Let's now call the procedure a couple of times to draw some triangles.

```
program Triangles;  
  procedure Tri (X, Y, Side : Integer);  
    begin  
      MoveTo(X, Y);  
      LineTo(X + Side, Y);  
      LineTo(X + Side div 2, Round(Y + Side / Sqrt(2)));  
      LineTo(X, Y)  
    end;  
  begin  
    Tri(10,10,10);  
    Tri(40, 40, 5);  
    Tri(80, 85, 18)  
  end.
```

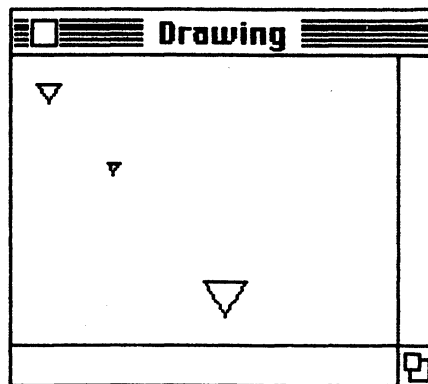


Figure 7-6.

## Exercises

1. Write a procedure that accepts an integer value and then prints the many blank lines in the text window.
2. What is printed by this program?

**program Example**

```
var
  A, B, X, Y : Integer;
procedure P1(A, B : Integer);
begin
  A := 3;
  B := 3;
end;
procedure P2(var A, B : Integer);
begin
  A := 3;
  B := 4;
end;
begin
  A := 2;
  B := 2;
  P1(X, Y);
  Writeln(X, Y);
  P2(X, Y);
  Writeln(X, Y);
  P2(Y, X);
  Writeln(X, Y)
end.
```

3. Write a procedure that returns the square of three numbers and a Boolean value of True if all three squares are even and False if all three are odd.

# 8 Arrays and Strings

Up to this point in the book the data types we have seen, Integer, Real, Char, Boolean, and Enumerated, are all known as scalar types. In a scalar type, each variable is a discrete entity. In this chapter, we will look at the data type Array which is a structured type. A structured type variable is a variable which can contain several distinct but related components. Also discussed in this chapter is the data type string which is a MacPascal extension.

Arrays are a structured type. Let's look at an example which will demonstrate the need for arrays. Let's write a program that will read and average three integers.

```
program Average;  
  var  
    A, B, C : Integer;  
    Avg : Real;  
  begin  
    Writeln('Enter three numbers to average');  
    Readln(A, B, C);  
    Avg := (A + B + C) / 3.0;  
    Writeln(Avg : 6 : 2)  
  end.
```

This is a straight forward problem and the solution is simple. However, how would a program be written to average 100 numbers, 1000, or 10000? It is obvious that it would be extremely impractical to use 100 separate variables to accomplish this. What is needed is an array.

An array is a group of variables of the same data type, all with a common name. Each individual variable (called an element) is referred by using a subscript along with the array name.



The declaration of an array is as follows:

**Num : array [1..10] of Integer;**

Index ↗
↖ Element type

Figure 8-1.

Here we have declared an array called Num. Num is a list of 10 memory locations, numbered 1 through 10, each holding an Integer. The standard form for declaring an array is:

**ArrayName : array [subrange] of DataType;**

ARRAY TYPE

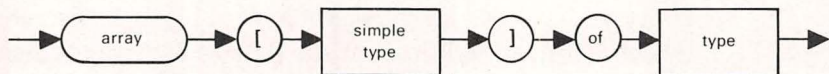


Figure 8-2. Syntax Diagram : Array Type.

An array is usually pictured as a linear list.

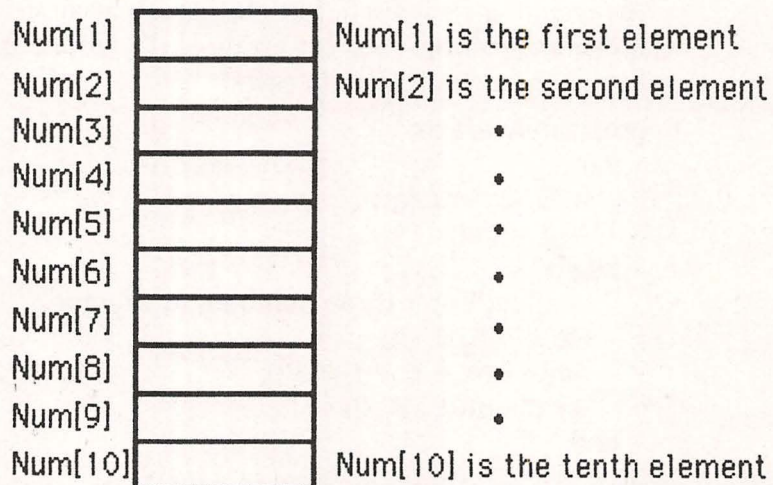


Figure 8-3.

The number of elements in an array and the range of subscripts is determined by the subrange used.

**A : array [3..7] of Integer;**

Array A has five elements A[3], . . . , A[7].

**B : array [-1..10] of Boolean;**

Array B has 12 Boolean elements, B[-1], B[0], B[1], . . . , B[10].

The subrange need not be Integer, it can be of any enumerated types.

**C : array [False,True] of Integer;**

Array C has two integer elements, C[False] and C[True].

**D : array ['A'..'Z'] of Boolean;**

Array D has 26 Boolean elements, D['A'], D['B'], . . . , D['Z'].

The type of array elements can be of any type, scalar or structured. We will limit the discussion here to scalar elements.

Each of the ten separate variables in an array are referred to by the array name and the subscript in brackets.

The subscript can either be a constant, or an expression which must evaluate to a legal subscript. The following examples all show legal subscripts.

```
Num[1] := 7;
Num[8] := Num[1] + 3;
K := 4;
Num[K] := 3;
Num[K + 2] := 4;
```

Arrays and for loops are a natural combination. The for loop can be used to sequentially access all the elements of an array. Let's use a for loop to initialize all the elements in array Num to zero.

```
for I := 1 to 10 do Num[I] := 0;
```

As this for loop iterates, the assignment statement is executed with all the different values of I. Thus, each of the ten elements in Num is set to zero. Let's now write the code to place into Num the following values (see Figure 8-4).

```
for I := 1 to 10 do Num[I] := I;
```

In this example, the values assigned to each element are the same as the subscript. Let's now, using an array, write the 100 number average program discussed in the beginning of the chapter.

```
program Average;
var
```



```
Sum, I : Integer;  
Avg : Real;  
Num : array [1..100] of Integer;  
begin  
  Sum := 0;  
  for I := 1 to 100 do  
    begin  
      Writeln('Enter number', I);  
      Readln(Num[I])  
    end; {For loop}  
  for I := 1 to 100 do  
    Sum := Sum + Num[I];  
  Avg := Sum / 100;  
  Writeln('The average is ', Avg :6 : 2)  
end.
```

The first **for** loop is used to read 100 values and place them into the array. The second **for** loop is used to add together all the values in the array. Finally, the average is computed. The use of the array has preserved the values entered for use in further calculations.

|         |    |
|---------|----|
| Num[1]  | 1  |
| Num[2]  | 2  |
| Num[3]  | 3  |
| Num[4]  | 4  |
| Num[5]  | 5  |
| Num[6]  | 6  |
| Num[7]  | 7  |
| Num[8]  | 8  |
| Num[9]  | 9  |
| Num[10] | 10 |

Figure 8-4.

## Sentinels

There is a limitation to the programming method used in the averaging program. The user of that program is forced to have to



enter 100 values in order for it to work. We can easily adapt the program to allow entry of any number of values to be averaged. The program can be redesigned to allow the user to enter any number of values up to 100. To mark the end of the data the user enters a negative number. The negative number is known as a sentinel, a value to be watched for, to signify the end of the data.

```

program NewAverage;
  var
    Int, Sum, I, Ct : Integer;
    Avg : Real;
    Num : array [1..100] of Integer;
begin
  Ct := 0;
  Writeln('Enter number to average');
  Readln(Int);
  while Int > 0 do
    begin
      Ct := Ct + 1;
      Num[Ct] := Int;
      Writeln('Enter number to average');
      Readln(Int)
    end;
    {Calculate average}
  Sum := 0;
  for I := 1 to Ct do
    Sum := Sum + Num[I];
  Avg := Sum / Ct;
  Writeln('The average is ',Avg : 6 : 2)
end.

```

In this version of the program a **while** loop is used to read values and to place them into the array. When the sentinel (negative number) is entered the condition in the **while** loop fails. The **for** loop then adds the contents of Num[I] through Num[Ct] and calculates the average. Once the average is calculated, we could easily find other statistical information since all the values entered are held in the array.

## Two-Dimensional Arrays

The data type of the elements in an array can also be an array.

**var**

**A : array [1..3] of array [1..2] of Integer;**

This creates a structure called a two-dimensional array. A two-dimensional array is pictured as a matrix. Here is the array declared above.

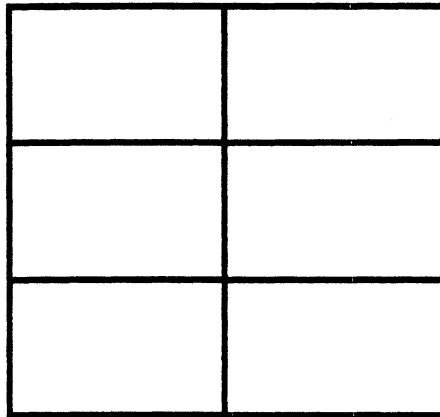


Figure 8-5.

This two-dimensional array is said to have three rows and two columns. Normally, we compact the declaration of a two dimensional array to:

**var**

**A : array [1..3,1..2] of Integer;**

The number of rows is always specified first. Each element in the array has two subscripts, one for the row and one for the column. Each element is referred to as: A[row,column]. The same array with each element marked with its subscript is shown in Figure 8-6.

Two-dimensional arrays are used to represent data that has a row and column relationship. A good example of this is a tic-tac-toe board.

Let's look at some examples involving two dimensional arrays.

|       | Column1 | Column 2 |
|-------|---------|----------|
| Row 1 | A[1,1]  | A[1,2]   |
| Row 2 | A[2,1]  | A[2,2]   |
| Row 3 | A[3,1]  | A[3,2]   |

Figure 8-6.

**var**

TwoD : array[1..3, 1..5] of Integer;

The above **var** section declares a two dimensional array of three rows and five columns. Let's fill each element in this array with it's column number.

```
for Row := 1 to 3 do
  for Col := 1 to 5 do
    TwoD[Row, Col] := Col;
```

The array would now appear as :

|   |   |   |   |   |
|---|---|---|---|---|
| 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 |
| 1 | 2 | 3 | 4 | 5 |

Figure 8-7.

Notice that the array is filled one row at a time. That is, the value of the outer for control variable, Row, was 1 while the values of the inner loops control variable, Col, varied from 1 to



5. The effect was the same as if the following statements had been executed.

```
TwoD[1, 1] := 1;
TwoD[1, 2] := 2;
TwoD[1, 3] := 3;
TwoD[1, 4] := 4;
TwoD[1, 5] := 5;
```

The same thing happens in the other rows.

Now let's write the code that will add together the value in a column of the array. This is done by holding a column constant as we vary the rows. Let's first add the first column.

```
Sum := 0;
for Row := 1 to 3 do
  Sum := Sum + TwoD[Row, 1];
```

This loop is equivalent to the following statement.

```
Sum := TwoD[1, 1] + TwoD[2, 1] + TwoD[3, 1];
```

Notice that these are the three elements in the first column. All the columns can be added together by placing this loop into another for loop.

```
for Col := 1 to 5 do
  begin
    Sum := 0;
    for Row := 1 to 3 do
      Sum := Sum + TwoD[Row, Col];
    Writeln('The sum of column', Col : 2, 'is', Sum : 2)
  end;
```

The contents of a row can be added together by holding the row constant while varying the columns.

```
Sum := 0;
for Col := 1 to 5 do
  Sum := Sum + TwoD[1, Col];
```

This is equivalent to the following statement.

```
Sum := TwoD[1, 1] + TwoD[1, 2] + TwoD[1, 3] +
      TwoD[1, 4] + TwoD[1, 5];
```

Now that we are familiar with using two dimensional arrays, we can write a program that acts as a tic-tac-toe board for a game

between two players. The program will keep track of the moves and inform the players of a win or tie. The tic-tac-toe board could be represented with a two-dimensional array with three rows and three columns. Each element in the array will hold an integer, 1 representing an X, and 0 representing an O. Here is the declaration of the array.

```
Board : array [1..3,1..3] of Integer;
```

Lets begin by writing the pseudocode for the program.

```
while no winner do
  begin
    Get player A's move
    Mark it in the array
    Print board
    Is it a win?
    Get player B's move
    Print board
    Is it a win?
  end
```

A second level of refinement includes some of the variable declarations, the procedure calls and the main program.

```
program TicTacToe;
  type
    Win = (Yes, No, Tie);
    Player = (A, B);
  var
    Board : array [1..3,1..3] of Integer;
    CurrentPlayer : Player;
    Winner : Win;
    Row, Column : 1..3;
    Sum, Total : Integer;
    .
    .
    .
  begin
    Winner := No;
    InitializeArray;
    CurrentPlayer := B;
```

```
while Winner = No do
  begin
    if CurrentPlayer = B then {Switch current player}
      CurrentPlayer := A
    else
      CurrentPlayer := B;
    GetMove;
    if CurrentPlayer = A then {Place move into array}
      Board[Row, Column] := 1
    else
      Board[Row, Column] := 0;
    PrintBoard;
    WinOrTie;
    if Winner = Yes then
      begin
        SysBeep(10);
        Writeln('Game won by ', CurrentPlayer)
      end; {If}
    if Winner = Tie then
      begin
        SysBeep(10);
        Writeln('Game ends in a Tie')
      end {If}
    end {While}
  end. {Game}
```

Now all that is left is to write the procedures.

The procedure `InitializeArray` will assign a -9 to all the elements in the array. This is done to make it easier to tell if there is a winner.

```
procedure InitializeArray;
  var
    R, C : Integer;
  begin
    for C := 1 to 3 do
      for R := 1 to 3 do
        Board[R, C] := -9
    end;
```

This procedure assigns a -9 to each element in the array. This is done with the help of two nested `for` loops. The loops will produce every combination of the subscripts in the following order.



```
Board[1, 1]
Board[2, 1]
Board[3, 1]
Board[1, 2]
Board[2, 2]
Board[3, 2]
Board[1, 3]
Board[2, 3]
Board[3, 3]
```

Note that the variable R is changed by the inner for loop which iterates three times for each value of C.

The procedure GetMove will prompt the current player for his move.

```
procedure GetMove;
begin
  Writeln('Player ', CurrentPlayer, ' Your Move');
  Writeln('Enter the row');
  Readln(Row);
  Writeln('Enter the column');
  Readln(Column)
end;
```

The procedure PrintBoard writes the two-dimensional array to the Text window. Two nested for loops are used for this.

```
procedure PrintBoard;
var
  R, C : Integer;
begin
  for R := 1 to 3 do
    begin
      for C := 1 to 3 do
        Write(Board{R, C}, ' ');
      Writeln
    end; {For R loop}
  Writeln
end;
```

The outer loop counts the rows, the inner loop the columns. This means the array is printed row 1 column 1, row 1 column 2, row 1 column 3, then row 2 column 1, etc. Note that after a complete row is printed a Writeln is done to advance to the next line.

Note also that the `Writeln` statement is not contained in the innermost loop.

The procedure `WinOrTie` will analyze the board and report back to the main program if there is a win or tie. To tell if there is a win we must check if either player has all three positions in a row, column, and diagonal of the array. Since player A's moves are marked by a one, if any row, column, or diagonal adds up to three then A is the winner. If any row, column, or diagonal adds up to zero then B is the winner. This is why the array elements were initialized to -9 rather than 0—so that three empty positions added together don't add up to zero.

```

procedure WinOrTie;
  var
    R, C : Integer;
begin
  Winner = No;
  if CurrentPlayer = A then
    Total := 3
  else
    Total := 0;
    {Check columns}
    for C := 1 to 3 do
      begin
        Sum := 0;
        for R := 1 to 3 do
          Sum := Sum + Board[R, C];
          if Sum = Total then
            Winner := Yes
        end; {for C loop}
      if Winner = No then
        begin
          for R := 1 to 3 do
            begin
              Sum := 0;
              {Check Rows}
              for C := 1 to 3 do
                Sum := Sum + Board[R, C];
                if Sum = Total then
                  Winner := Yes
              end {For R loop}
            end; {If}

```

```

if Winner = No then
  begin
    {Check diagonals}
    Sum := 0;
    for C := 1 to 3 do
      Sum := Sum + Board[C, C];
    if Sum = Total then
      Winner := Yes
    end; {If}
  if Winner = No then
    begin
      Sum := 0;
      Winner := Tie;
      {Look for any empty position}
      for C := 1 to 3 do
        for R := 1 to 3 do
          if Board[R, C] = -9 then
            Winner := No
          end {If}
        end; {WinOrTie}
    end; {WinOrTie}
  end; {WinOrTie}

```

The output of the routine is in the global variable Winner. Depending upon the move of the last player, Winner is set to either Yes, No, or Tie, the three possible values of the data type Win. This routine is divided into three parts. First all columns are checked. We add together the contents of each of the columns. This is done with two for loops. The column which is varied by the outer loop is held constant as the inner loop controls the rows. Once all three elements in a column are added the sum is compared to the total which would indicate a winning move was completed for that player. All three columns are checked regardless if a winner has been determined already. This is because the rows are checked with for loops and there is no way to abort the checking even after finding the condition we want (this could have been done with a while loop). However this is no problem since Winner is initially set to No and only set to Yes if a winning row is found. A variable used in this way is called a toggle or switch.

```

  {Check columns}
  for C := 1 to 3 do
    begin
      Sum := 0;

```



```
    for R := 1 to 3 do
        Sum := Sum + Board[R, C];
    if Sum = Total then Winner := Yes;
end; {For Loop}
```

If no winning column is found we now move on to check the rows. The rows are checked in the same manner except that the columns are kept constant as the loop adds together the contents of each row.

```
{Check rows}
for R := 1 to 3 do
    begin
        Sum := 0;
        for C := 1 to 3 do
            Sum := Sum + Board[R, C];
        if Sum = Total then
            Winner := Yes;
        end; {For}
```

The diagonals have to be checked in a different way. The major diagonal, the one that goes from the upper left hand corner to the lower right hand corner consists of the elements: Board[1, 1], Board[2, 2], and Board[3, 3]. Note that in each of these elements the row and column subscripts are the same. The contents of these elements can be added with one for loop using the loop control variable as both subscripts.

```
for C := 1 to 3 do
    Sum := Sum + Board[C, C];
if Sum = Total then
    Winner := Yes;
```

The minor diagonal, the one that goes from the upper right hand corner to the lower left hand corner, contains the elements: Board[1, 3], Board[2, 2], and Board[3, 1]. All these elements can be added with one for loop if the relationship between the row and column subscripts is noticed.

```
for C := 1 to 3 do
    Sum := Sum + Board[C, 4 - C];
if Sum = Total then
    Winner := Yes;
```

If no winning move was made the array is now checked to see if there is a tie. In tic-tac-toe there is a tie if there is no place for a

player to move. This would be represented in the array by no element equal to the initial value of -9. First, Winner is set to Tie and then two nested for loops are used to examine the contents of all the array elements. If any element is found to have the initial value then there is a place to move and Winner is toggled to No.

```

if Winner = No then
  begin
    Sum := 0;
    Winner := Tie;
    {Look for any empty position}
    for C := 1 to 3 do
      for R := 1 to 3 do
        if Board[R, C] = -9 then
          Winner := No
  end; {If}

```

Here is the TicTacToe program all together. This program has more than just entertainment value. Running the program we help acquaint you with the row and column positions in a two dimensional array. Worth noting in this program listing is the comment line of dashes used to separate procedures. This is done to improve the readability of the program.

```

program TicTacToe;
  type
    Win = (Yes, No, Tie);
    Player = (A, B);
  var
    Board : array[1..3, 1..3] of Integer;
    CurrentPlayer : Player;
    Winner : Win;
    Row, Column : 1..3;
    Sum, Total : Integer;
  procedure InitializeArray;
    var
      R, C : Integer;
    begin
      for C := 1 to 3 do
        for R := 1 to 3 do
          Board[R, C] := -9;
    end; {Initialize Board}
  {-----}

```

```
procedure GetMove;
begin
  Writeln('Player ', CurrentPlayer, ' Your Move');
  Writeln('Enter the row');
  Readln(Row);
  Writeln('Enter the column');
  Readln(Column)
end; {GetMove}
{-----}
procedure PrintBoard;
var
  R, C : Integer;
begin
  for R := 1 to 3 do
    begin
      for C := 1 to 3 do
        Write(Board[R, C], ' ');
      Writeln
    end; {For R loop}
  Writeln
end;
{-----}
procedure WinOrTie;
var
  R, C : Integer;
begin
  Winner := No;
  if CurrentPlayer = A then {Switch current player}
    Total := 3
  else
    Total := 0;
  {Check columns}
  for C := 1 to 3 do
    begin
      Sum := 0;
      for R := 1 to 3 do
        Sum := Sum + Board[R, C];
      if Sum = Total then
        Winner := Yes
    end; {For C loop}
  {Check Rows}
  if Winner = No then
```



```

begin
  for R := 1 to 3 do
    begin
      Sum := 0;
      for C := 1 to 3 do
        Sum := Sum + Board[R, C];
        if Sum = Total then
          Winner := Yes
        end {For R loop}
      end;
    if Winner = No then
      begin
        {Check diagonals}
        Sum := 0;
        for C := 1 to 3 do
          Sum := Sum + Board[C, C];
          if Sum = Total then
            Winner := Yes
          end; [IF]
        if Winner = No then
          begin
            Sum := 0;
            Winner := Tie;
            {Look for any empty position}
            for C := 1 to 3 do
              for R := 1 to 3 do
                if Board[R, C] = -9 then
                  Winner := No
                end
              end;
            end;
          }
        begin {Main program}
          Winner := No;
          InitializeArray;
          CurrentPlayer := B;
          while Winner = No do
            begin
              if CurrentPlayer = B then
                CurrentPlayer := A
              else
                Board[Row,Column] := 0;
                CurrentPlayer := B;

```

```
GetMove;
if CurrentPlayer = A then
  Board[Row, Column] := 1
else
  Board[Row, Column] := 0;
PrintBoard;
WinOrTie;
if Winner = Yes then
  begin
    SysBeep(10);
    Writeln('Game won by ', CurrentPlayer)
  end; {If}
if Winner = Tie then
  begin
    SysBeep(10);
    Writeln('Game ends in a Tie')
  end {If}
end {While loop}
end. {TicTacToe}
```

An interesting way to adapt the TicTacToe program is to replace the two dimensional **array** of Integer with a two dimensional **array** of a user defined type with three values, a value of each of the two player's moves and a third value for an unused position (initial value). Think about the changes that would be required in the WinOrTie procedure.

A second and more complex change in the program is to replace one of the players with the computer itself. This requires the program to identify which of the empty positions is the most advantageous and requires analysis of the strategy of the TicTacToe game. To proceed, play several games on paper and try to identify why you made each move. Then try to quantify your reasoning into an algorithm which can be programmed.

## Arrays of Characters

A limitation of standard Pascal is that the type Char can only hold a single character. An array whose elements are of type Char can be used to handle a stream of character data. To read from the keyboard and store up to 80 characters we can declare an array as follows:

```
Inchar : array [1..80] of Char;
```

The array Inchar has 80 elements each one capable of holding one character. We can read characters from the keyboard and place them into Inchar with:

```

I := 0;
Writeln('Enter a message');
Read(Ch);
while not (EOLN) do
  begin
    I := I + 1 ;
    Inchar[I] := Ch;
    Read(Ch);
  end; {while}

```

We want this loop to keep on reading characters from the keyboard until a carriage return is entered. EOLN, which stands for End Of Line is helpful in such situations. EOLN is a special Boolean function given a value by MacPascal. There is no need to declare it in your program. EOLN has a value of False if a carriage return has not been entered and True if a carriage return has been entered. We want the loop to continue executing when EOLN is False, but a While loop executes when the condition is True. We therefore reverse the value of EOLN with the use of a **not**. When EOLN is used, the Read statement must be used instead of Readln. Using Readln will cause the program to work improperly. Note that in this loop, the characters are first read into the Char variable Ch and then placed into the array. This is done to trap the carriage return character and not place it into the array.

This loop could also be written a different way without using EOLN. We could use the while loop to check the ASCII value of the character entered.

```

I := 1;
Writeln('Enter a message');
Read(Ch);
while ORD(Ch) < > 13 do
  begin
    Inchar[I] := Ch;
    I := I + 1;
  end;

```



```
    Read(Ch)
  end; {While}
```

Note that the ASCII code for a carriage return is 13.

## The Code Breaker Program

Secret codes are often broken by examining the frequency that the characters appear in a message and comparing that to a list of the frequency of characters in unscrambled English. We can count the frequency of characters appearing in a message by expanding on the routine just developed.

First, let's declare a second array.

```
Freq : array [' '..'z'] of Integer;
```

The subrange contains all the characters whose ASCII codes are from 32 to 122, or the 91 characters from a space to a lower case 'z'. The array therefore contains 91 elements, each one containing an integer. Don't confuse this array which uses a character as a subscript but contains an integer with an array whose elements are of type Char. We can now step through the Inchar array and increment the element in Freq whose subscript is the same as the character being examined.

```
program CodeBreaker;
var
  Inchar : array [1..80] of char;
  Freq : array [' '..'z'] of Integer;
  Ch : Char;
  I, NumOfChar : Integer;
begin
  for Ch := ' ' to 'z' do
    Freq[Ch] := 0; {Initialize array}
  {Read in message}
  I := 0;
  Writeln('Enter the message');
  Read(Ch);
  while not (eoln) do
    begin
      I := I + 1;
      Inchar[I] := Ch;
      Read(Ch)
    end; {while}
```

```

NumOfChar := I;
I := 1;
{Count frequency of characters}
while I <= NumOfChar do
  begin
    Ch:=Inchar[I];
    Freq[Ch] := Freq[Ch] + 1;
    I := I + 1
  end;
{Write frequencies}
for Ch := ' ' to 'z' do
  Writeln('The number of ', Ch, ' are', Freq[Ch]: 2)
end.

```

Note that when counting the frequencies the Char variable Ch is used to hold a character from Inchar and then is used as the subscript of Freq.

## Strings

Character data is often processed by a computer. However, handling character data in an array whose elements are of type Char is inefficient and awkward. Fortunately, MacPascal has available an extended data type known as **string**. A variable of type **string** can hold a sequence of characters. The sequence can be from 1 to 255 characters long. **string** has some of the attributes of a scalar type and some of the attributes of a structured type and is therefore considered to be neither. A variable is declared to be a string with:

```
S : string[80];
```

This declares S to be a string variable. MacPascal boldfaces the word **string** just like it does **array**. The maximum number of characters the string can hold is called the size and is indicated in brackets. The default size is 255 if no size is specified. The size of S is 80, but the length of the string may dynamically vary from 0 to 80 characters at any time. We could say that a string's size is its maximum length. A value is placed in a **string** variable with an assignment statement.

```
S := 'ABCD';
```

The string value 'ABCD' is enclosed in single quotes. The current length of string S is 4. A string variable can be cleared by

assigning to it the null string. The null string is a string with no contents indicated by two consecutive quotes. The length of the null string is zero.

```
S := ""; {assign null string to S}
```

An error will occur if an attempt is made to assign a string value whose length is greater than the size of the string variable. For example, the following program segment will produce a run-time error.

```
var  
  S1 : string[4];  
.  
.  
S1 := 'ABCDEF';
```

The individual characters in a string can be accessed as though they were in an array. For instance,

```
S := 'ABCD';  
Write(S[1]);
```

would print an 'A'. The integer in the brackets is an index to the characters in the string. An attempt to reference S[0] or a position greater than the current length of the string would result in a run-time error. The contents of a string can also be changed in this manner.

```
S := 'HI';  
S[1] := 'B';  
WriteLn(S);
```

The above program segment would print BI.

## Reading a String

String values can be read from the keyboard using ReadLn.

```
ReadLn(S1, S2);
```

This readLn statement will read two strings, S1 and S2. A carriage return is used as a sentinel to signify the end of a string when reading it from the keyboard. The carriage return is not placed in the string.



## Comparing Strings

The value of two strings can be compared in a Boolean expression. The comparison is based on the value of the ASCII codes of the characters in the strings. When two strings of different lengths are compared, each character in the longer string is considered to be greater than the missing characters in the shorter strings. Two strings have to be of equal length to be equal.

'AB' is greater than 'AA'  
 'AT' is less than 'ATTACH'  
 'BILL' equals 'BILL'  
 'BILL' is not equal to 'HILL'

Numerically, the ASCII codes for upper case letters is smaller than lower case letters, so that

'a' is greater than 'A'

An array of strings can be alphabetized in this way.

## The String Functions and Procedures

The arithmetic operators (+, -, /, div and mod) can not be used with string values. Operations on strings are performed with the help of a set of built-in functions and procedures.

### The Length Function Length(string)

Length returns the current length of the string specified.

```
S:= 'GOOD';  
Write(Length(S));
```

prints 4.

### The Concat Function Concat(S1, S2, . . . )

The Concat function is used to combine any number of strings into one.

```
S:=Concat('GOOD', ' ', 'MORNING');
```

The value of S is now 'GOOD MORNING'. The length of the result can not exceed 255 or else a run-time error will occur.

## The Pos Function   Pos(Substring, String)

The Position function returns the position of the first occurrence of the substring in the string as an integer.

```
I := Pos('CD', 'ABCD');
```

assigns 3 to I, since the substring 'CD' starts at the third position in 'ABCD'. If the substring is not present a zero is returned.

## The Copy Function   Copy(String, Index, Count);

The Copy function returns a string of count characters starting at String[Index].

```
StrgVar := Copy('ABCDE', 3, 2)
```

assigns to StrgVar

```
'CD'
```

## The Delete Procedure   Delete(String, Index, Count)

The Delete procedure removes from the specified string Count number of characters starting at String[Index].

```
S := 'ABCDE';  
Delete(S, 3, 2);
```

results in the value of S being 'ABC'. If characters outside the length of the string are referenced it is not an error. Only the characters that lie within the range are deleted. Note that Delete is a procedure not a function and is used as a statement.

## The Omit Function   Omit(String, Index, Count)

The Omit function is similar to the Delete procedure except that the value of the string is not changed. Instead the new string is returned as the value of the function.

```
StrgVar := Omit('ABCDE', 2, 2);
```

assigns to StrgVar:

```
'ADE'
```

## The Insert Procedure    **Insert(Source, Destination, Index)**

The Insert procedure places the destination **string** into the source **string** at the index position.

```
S := 'ABCDE';
Insert(S, 'FF', 3);
```

The value of S is now

ABFFCDE

## The Include Function    **Include(Source, Destination, Index)**

The Include function is similar to the Insert procedure except that the **string** is not affected. Instead the value of the new **string** is returned as the value of the function.

```
StrgVar := Include('ABCDE', 'FF', 3);
```

assigns to StrgVar:

'ABFFCDE'

## Exercises

1. Declare an array for each of the following. How many elements are in each array?
  - a. An Integer **array** where the index goes from 2 to 20.
  - b. A Real **array** where the index goes from -4 to 50.
  - c. A Boolean **array** where the index goes from -100 to 201.
2. How many elements are in the following **arrays**?
  - a. **array** [-30..30] of Char
  - b. **array** [1..10, 1..3] of Real
  - c. **array** [2..12, 1..3] of Real
  - d. **array** ['A'..'F'] of Integer
3. Write a procedure that finds the largest value in an **array** of integers.



4. Write a procedure that reverses the elements in the following **array**. The last becomes the first, etc.

**A : array [1..10] of Integer;**

5. Set all the elements in the following **array** to -5.

**B : array [1..5, 1..4] of Integer;**

6. Suppose the following **array**

**Board array [1..8, 1..8] of Boolean;**

is used to represent a chessboard. An element (corresponding to a position on the board) set to True represents a queen in that position. Write a program that positions eight queens on the chessboard such that no two queens are in the same row, column, or diagonal. Try to produce several solutions.

7. Write a procedure that accepts a **string** and then prints it reversed.
8. Write a procedure that converts the contents of an **array** of characters into a single **string**.
9. Write a procedure that continually accepts input of single characters and places them into a **string**. The input stream ends when a '+' or '-' is entered.
10. Adapt the procedure described in exercise 9 to make sure that no more than 255 characters are placed into the **string**.

# 9 More On Structures

In Chapter 4, we were introduced to some of Pascal's structures. In this chapter we will expand our "Pascal vocabulary" with a new loop and several new structures.

## The Repeat Loop

The third of Pascal's loop structures is the **repeat** loop. **repeat** is a free loop (like **while**, unlike **for**) and can be roughly described as an upsidedown **while** loop.

The form of the **repeat** loop is:

```
repeat  
  statements  
until expression is TRUE;
```

The execution of a **repeat** loop is as follows. First, the statements included in the loop are performed. Then the condition is checked, if it is FALSE the loop is repeated, otherwise, it is terminated.

The differences between the **repeat** and the **while** loops are:

1. The **repeat** checks the condition at the bottom of the loop rather than at the top as is done in the **while** loop, so in the **repeat** loop, the body of the loop is always executed at least once. In the **While** loop, if the condition is initially false, the body of the loop will not be executed at all.
2. The **repeat** and **Until** automatically bracket a compound statement. No **begin** and **end** are needed.
3. The **repeat** loop iterates if the condition is false, the **while** loop iterates when the condition is true.



This is the flowchart for the **repeat** loop.

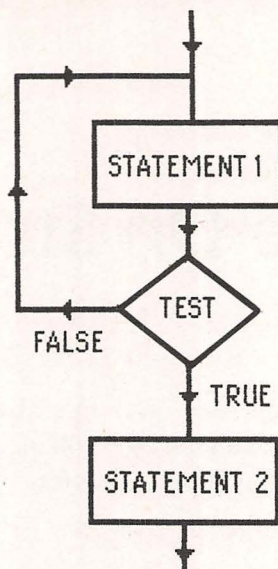


Figure 9-1.

The syntax for the **repeat** loop is shown in the next diagram

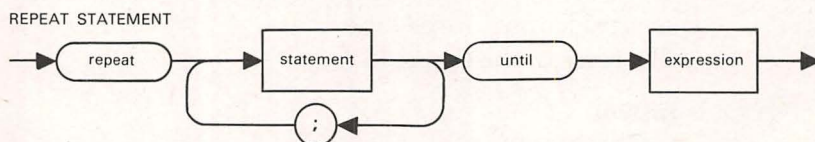


Figure 9-2. Syntax Diagram : **repeat** Statement.

Let's look at a **repeat** loop in detail side by side with a **while** loop.

```

I := 1;
repeat
  Writeln(I);
  I := I+1
until I > 10;
  
```

```

I := 1;
while I <= 10 do
  begin
    Writeln(1);
    I := I+1
  end;
  
```

These loops both print the integers from 1 to 10. In examining the differences notice where in the loop the condition is checked and how the opposite conditions are used. The **repeat** loop uses the condition  $I > 10$  and the **while** loop uses the condition  $I \leq 10$ .



A useful application of the **repeat** loop is to check the validity of input. For example, a program that analyzed exam grades might use the following loop to make sure that the values entered as exam grades were valid.

```
repeat
  Writeln('Enter Grade');
  Readln(Grade)
until (Grade > 0 and Grade <= 100);
```

Here the condition is a multiple test. Since we want the value entered to fall inside a range, the logical operator **and** is used. If the value entered does not fall within the range, the condition is not met and the loop is done again. Think of the difference between this and declaring the variable `Grade` as the subrange 1..100. In the case of using the subrange, if a value out of the range is entered, an error would occur while the program was executing stopping the program. This loop would not stop execution but rather would prompt the user to enter the value a second time. This method is much preferable since a major goal in programming is to prevent a program from crashing (stopping due to an error) from run time errors.

## The Bubble Sort

A sort is an algorithm used to place the values in an array in numerical order. There are many different sorting methods, the bubble sort being one of the easiest to follow and sufficiently efficient to be used in many situations. A book on data structures will explain in much more detail different sorts and the criteria used to evaluate them.

The bubble sort operates by comparing each two adjacent elements in an array. If they are out of order they are exchanged. The comparing runs through the entire array. After a pass is complete, if any exchanges were made, the process is started again at the top. When no exchanges take place in a pass through the array, then the array is sorted.

Here is the bubble sort in a procedure. The array to be sorted is passed to the procedure in a variable parameter and is of the global type `ArrayType`. Its elements are of the global type `ElementType`. The parameter `NumElements` is an integer containing the number of elements in the array.

```

procedure BubbleSort(var A : ArrayType; NumElements :
    Integer);
type
    Exch = (Yes, No);
var
    Exchanged : Exch;
    Temp : ElementType;
    I : Integer;
begin
    repeat
        I := 1;
        Exchanged := No;
        for I := 1 to NumElements - 1 do
            if A[I] < A[I+1] then
                begin
                    Exchanged := Yes;
                    Temp := A[I+1];
                    A[I+1] := A[I];
                    A[I] := Temp;
                end
            until Exchanged = No
        end; {procedure}

```

|   |
|---|
| 4 |
| 1 |
| 3 |
| 2 |
| 5 |

BEFORE

|   |
|---|
| 1 |
| 3 |
| 2 |
| 4 |
| 5 |

AFTER PASS 1

|   |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |

AFTER PASS 2

Figure 9-3.

## The Case Statement

The case statement can be used to replace several if statements. The following nested if statements:

```

if I = 1 then
    WriteLn('ONE')

```

```

else if I = 2 then
    Writeln('TWO')
    else if I = 3 then
        Writeln('THREE')
    else
        Writeln('NONE OF THESE');

```

can be replaced with:

```

case I of
    1:
        Writeln('ONE');
    2:
        Writeln('TWO');
    3:
        Writeln('THREE');
    otherwise
        Writeln('NONE OF THESE')
end; {CASE}

```

In the **case** statement, the key word **case** is followed by an expression whose value may be any ordinal type except **LongInt**. This is followed by the key word **of**. Between the **of** and the **end** statements are a list of statements each labeled by a value of the same type as the expression. When the **case** statement is executed the expression is evaluated. The statement whose label matches the value of the expression is then executed. Since an expression can only have one value, only one of the statements in the **case** is executed. This means all the statements are executed mutually exclusively. If the value of the expression is not found the statement following the key word **otherwise** is executed. If in this situation the **otherwise** clause is absent (it is not required) then an error occurs. **otherwise** is an extension not found in standard Pascal.

More than one value may be used as a label as shown in the next example.

```

case I of
    1, 2, 3:
        Writeln('1 to 3');
    4, 5, 6:
        Writeln('4 to 6')
end;

```



Here is the syntax diagram for the **case** statement:

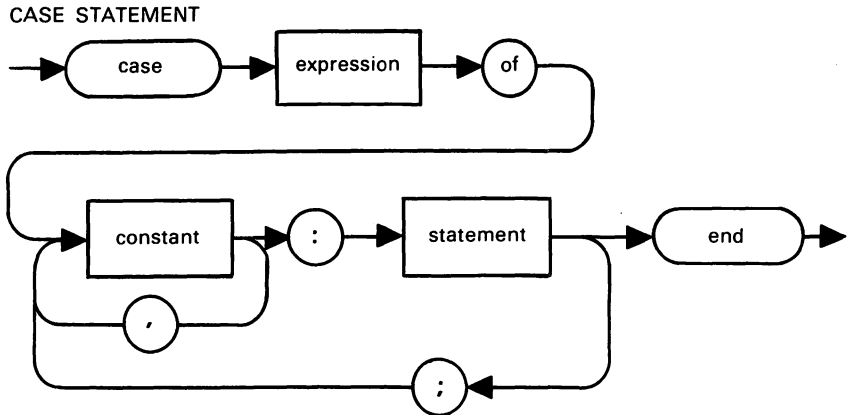


Figure 9-4. Syntax Diagram : **case** Statement.

If the value of *I* is from 1 to 3 then the first statement is executed. If it is from 4 to 6 then the second statement is executed.

Case statements are useful in menu applications. A menu is a choice of possible program options for the program user to choose from. These are menus your MacPascal program will display in the Text window, not the pull-down menus you are familiar with from working with MacPascal. First, the menu choices are displayed, the user enters a selection and a **case** statement is used to branch to a procedure based upon the choice entered. Here is a menu from a data management program.

```

Writeln('1 for Add data');
Writeln('2 for List data');
Writeln('3 for Print data');
repeat
  Readln(N)
until N >= 1 AND N <= 3;
case N of
  1 :
    begin
      Writeln('Add selected');
      Add {Add procedure}
    end;
  2 :
    begin
      Writeln('List selected');

```

```

        List {List procedure}
    end;
3:
    begin
        Writeln('Print selected');
        Print {Print procedure}
    end
end; {case statement}

```

Note that no **otherwise** clause is used. This is because there is no danger of a program error due to invalid input, since the **repeat** loop makes sure that the value entered falls within the range examined by the **case** statement. Also note that compound statements are used as the case options.

## User-Defined Functions

User-defined functions are the second type of subprograms in Pascal. They are used when only a single value needs to be returned from a subprogram. They differ from procedures only in the way that they are called and how information is passed back to the calling routine.

A procedure is called by using its name as a statement. Information is sent to a procedure with the use of value and variable parameters and global variables. Information is returned from a procedure with either variable parameters or global variables. On the other hand, a function is called by using its name like a variable. Information is sent to a function using either value parameters or global variables. Only a single value can be returned from a function and this is done by assigning the value to the name of the function.

The form of a function is very similar to a procedure except that the function heading is slightly different.

```
function Power(Base, Exponent : 1..MaxInt) : Integer;
```

Here the function **Power** is declared with two value parameters. Notice the key difference between this and a procedure heading.

```
procedure Power(Ans, Power, Exponent : 1..MaxInt);
```

In the function heading a type declaration is included to declare

the type of the value returned by the function. This is placed following a colon after the parameters.

```

var
  I, Ans : Integer;
begin
  Ans := 1;
  for I := 1 to Exponent do Ans := Ans * Base;
  Power := Ans
end;{function}

```

Here is the body of the function. It is analogous to the body of a procedure. There is however one difference. The result of the function (remember a function returns only one value) is assigned to the function name (Power). The function name acts like a bridge between the function and the statement that calls it holding the value to be returned. Other values can be returned via global variables or variable parameters.

A function is called by using the function name as a variable. The value of the function eventually replaces the function name after the function has executed.

```
X := Power(3, 4)
```

This call to the function Power will assign a value of 81 ( $3^4$ ) to the variable X. The value returned by Power is an integer because of the type declaration we used in the function heading. All the rules for assignment hold for the values returned by functions. Notice how the use of a user-defined function is the same as the use of a pre-defined function. This is because they are the same, except that the pre-defined functions have already been written and are held in the MacPascal interpreter ready for use. For example, the pre-defined function ODD might be written like this:

```

function Odd(N : Integer) : Boolean
begin
  if (N div 2) * 2 = N then
    Odd := FALSE
  else
    Odd := TRUE;
  end; {Odd}

```

Remember that in integer division the fraction value is lost. When  $N \text{ div } 2$  is done the remainder is lost if N is odd. For



instance,  $5 \text{ div } 2$  equals 2. When the result is multiplied by 2 it no longer equals the original odd value. This is not true for an even number.

Here is a rundown on the differences between functions and procedures.

|                 | Functions  | Procedures  |
|-----------------|--|---|
| Called by:      | Use name like variable   | Use name as statement                             |
| Value returned  | One as function name, others as variable parameters and global variables | Many via variable parameters and global variables |
| Parameters Used | Value  | Value and variable                                |

## Recursion

An interesting question is what happens when a subprogram calls itself. The result is not a loop but a very powerful and complicated technique called recursion. Recursion occurs when a function or procedure invokes itself. Recursion can be easily demonstrated by writing a function that calculates the factorial of a number. The factorial of a number, denoted with an exclamation point such as  $N!$ , is that number multiplied by all its predecessors down to one. For instance,  $5!$  is  $5*4*3*2*1$  which equals 120. Factorials are used extensively in statistical and probabilistic work. We can also define the factorial of a number as that number itself times the factorial of that number minus one or  $5! = 5*4!$ . This is a recursive definition. A recursive definition is not defined in terms of itself as it might appear but rather is defined in a simpler version of itself. Here is the function.

```

function Factorial(N : Integer) : Integer;
begin
  if N = 0 then
    Factorial := 1
  else
    Factorial := N * Factorial(N - 1)
end; {factorial}

```



This function has just one statement. If the value of  $N$  is zero, then the function simply returns a one and ends. If  $N$  is greater than one, say 5, then the `else` clause is done. This is where the recursive definition takes place.  $N$  factorial is defined as  $N$  times  $N-1$  factorial. At this point what happens tends to be confusing but need not be. The function now calls the function `Factorial` which happens to be itself. The current state of the function (the value of the variables and which statement is executed in the function call) is saved and the function is started again with the new parameter. When this second call is terminated, the original call to the function is resumed with the value returned by the second call. Where this starts to get confusing is when a second call to the function calls the function again and so on. This is similar to a busy executive on the phone. When a second call comes in, the executive puts the first caller on hold and attends to the second caller. When a third call comes the second caller also goes on hold. Depending upon how busy she is this sequence of events may take place several times. Of course, in order for this not to go on forever there must be some mechanism to end a call. Eventually, a conversation ends and the next to the last call is resumed (assuming, of course, that the person on the line was patient). The process repeats until she is back to the original call.

This is also true of recursive functions, there must be some way for each call to the function to terminate. We call this the stop rule. In this procedure the stop rule is : after  $N$  calls to the function, the value of the parameter  $N$  will be 0. This will cause the **then** part of the `if` statement to execute and that call to the function to finish returning a value. If there was no mechanism for the recursive calls to end, the mechanism used to track the successive calls to the procedure would overflow and cause a run time error. When that call ends the function that called it is now provided with a value and can also end. Recursive calls to a function are often viewed as levels, not unlike several windows on the Macintosh's screen sitting on top of each other. When the top window closes the one under it becomes active. Figure 9-5 demonstrates the recursion process.

Demonstrated in the diagram is the successive calls to the function `Fact` for the value of 5 and the value of the parameters for each call. Notice that no value is returned until the fifth call to the function. At that point each previous call is evaluated and returns to the call under it.

Graphics are a further way to help demonstrate recursion. A

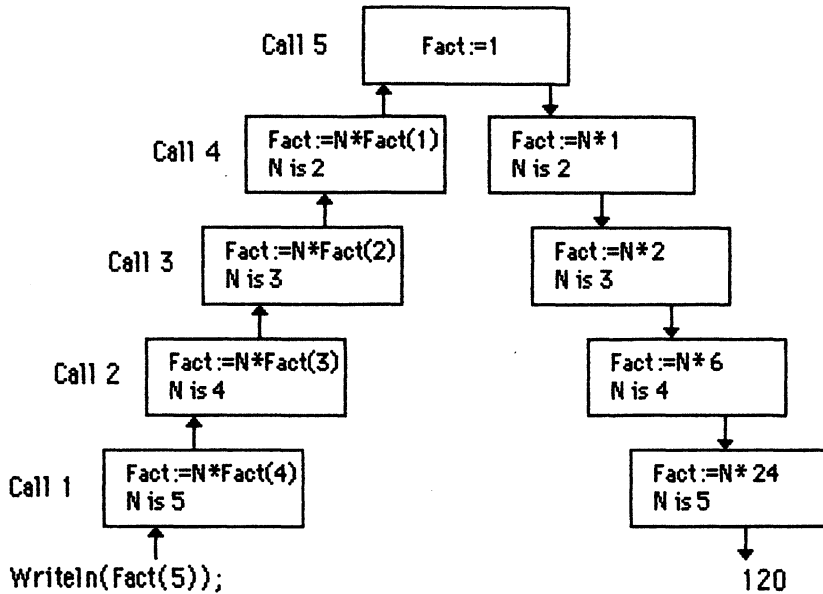


Figure 9-5.

simple program with rectangles can help show the levels of nesting involved in recursive calls.

The following program calls a procedure that draws a rectangle. That procedure draws a rectangle and then recursively calls itself to draw a smaller rectangle inside of the one previously drawn. The stop rule will check to see that the rectangle to be drawn can't be drawn (the left and right side of the rectangle overlap).

```

procedure Recurse;
  var
    R : Rect;
  procedure Box (U, D : Integer);
  begin
    if (D - U) > 0 then
      begin
        SetRect(R, U, U, D, D);
        FrameRect(R);
        Box(U + 5, D - 5); {Recursive call}
      end
    end;
  begin
    Box(1, 150)
  end.
  
```



When the program is run 16 recursive squares will appear in the **Drawing** window.

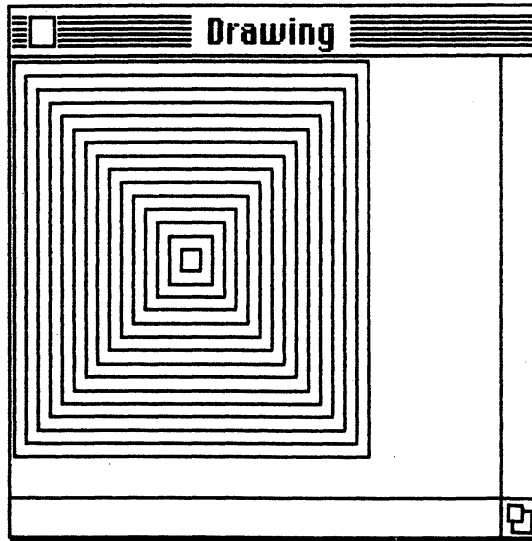


Figure 9-6.

One of the unique features of MacPascal, concurrently open windows, can be used to help keep track of the procedure calls. As we display the boxes in the **Drawing** window, we can also display a message in the **Text** window right before a recursive call to the procedure. We can display a second message on returning from a recursive call. A new variable can be added to keep track of the number of recursive calls.

```
procedure Recurse;
var
    I : Integer;
    R : Rect;
procedure Box (U, D, It: Integer);
begin
    if (D - U) > 0 then
        begin
            SetRect(R, U, U, D, D);
            FrameRect(R);
            Writeln('Call it', It);
            Box(U + 5, D - 5, It + 1); {Recursive call}
        end;
        Writeln('Returning from', It)
```

```

    end;
begin
    I := 0;
    Box(1, 150, I)
end.

```

This procedure draws a rectangle prior to a recursive call to itself and thus draws smaller rectangles inside of bigger. We can reverse the drawing order (bigger outside of smaller) by moving the `FrameRect` statement to after the recursive procedure call. This way no rectangle is drawn to after the stop rule is reached and therefore the smallest rectangle is drawn first. Another interesting variation in the program is to keep the `FrameRect` in the original position and then add an `EraseRect` statement after the stop rule. This will display all the rectangles prior to reaching the stop rule and erasing them after the stop rule is reached.

While the two recursion examples we have seen are interesting and intellectually stimulating they none the less demonstrated programs that could have been written easily without recursion. The next two examples harness the power of recursion to simplify more complex tasks.

Let's look at a program that displays a large square and then recursively divides itself into four smaller squares. Each rectangle will continue to divide itself until the stop rule is hit. In this case, the stop rule tests to see if the width of a square is less than five points.

```

program SubDivide;
  procedure Box (Left, Top, Right, Bottom : Integer);
  var
    Hcenter, Vcenter : Integer;
  begin
    if (Right - Left) >= 5 then
      begin
        FrameRect(Left, Top, Right, Bottom);
        Hcenter := (Left + Right) div 2;
        Vcenter := (Top + Bottom) div 2;
        Box(Left, Top, Hcenter, Vcenter);
        Box(Hcenter, Top, Right, Vcenter);
        Box(Left, Vcenter, Hcenter, Bottom);
        Box(Hcenter, Vcenter, Right, Bottom)
      end
    end;
end;

```

```

begin
  Box(0, 0, 256, 256)
end.

```

This program differs from the one before because there are four recursive calls to the procedure rather than one. The original call to the procedure from the main program draws the largest box and then calls itself with the first recursive call. This new call to the procedure draws a box in the upper left hand corner of the largest box and then calls itself again. This new call draws a box in the upper left hand corner and the recursive calls continue until the stop rule is enacted. At this point all the upper left hand corner boxes have been drawn and the last recursive call returns to the second recursive call, which starts drawing upper right hand corner boxes.

The snapshot of the **Drawing** window (Figure 9-7) shows the program in progress. Run the program and watch the order in which the boxes are drawn. Also try running the program with **Step** to observe the order of the recursive calls. Move the **FrameRect** statement to after the **procedure** call and watch the order in which the boxes will then be drawn.

Our last recursion example utilizes the procedure written in Chapter 7 to draw equilateral triangles.

```

program Recursive __ Triangles;
  procedure Tri (X, Y, Side : Real);
  begin
    if Side >= 5 then
      begin
        Tri(X, Y, Side / 2);
        Tri(X + Side / 2, Y, Side / 2);
        (X + Side / 4, Y + Side / Sqrt(2) / 2, Side / 2);
        Move To (Round(X), Round(Y));
        LineTo(Round(X + Side), Round(Y));
        LineTo(Round(X + Side / 2), Round(Y + Side / Sqrt(2)));
        LineTo(Round(X), Round(Y));
      end
    end;
  begin
    Tri(10, 10, 300)
  end.

```

This procedure differs slightly from the one written in Chapter 7. Real values are used instead of integers to increase the accu-



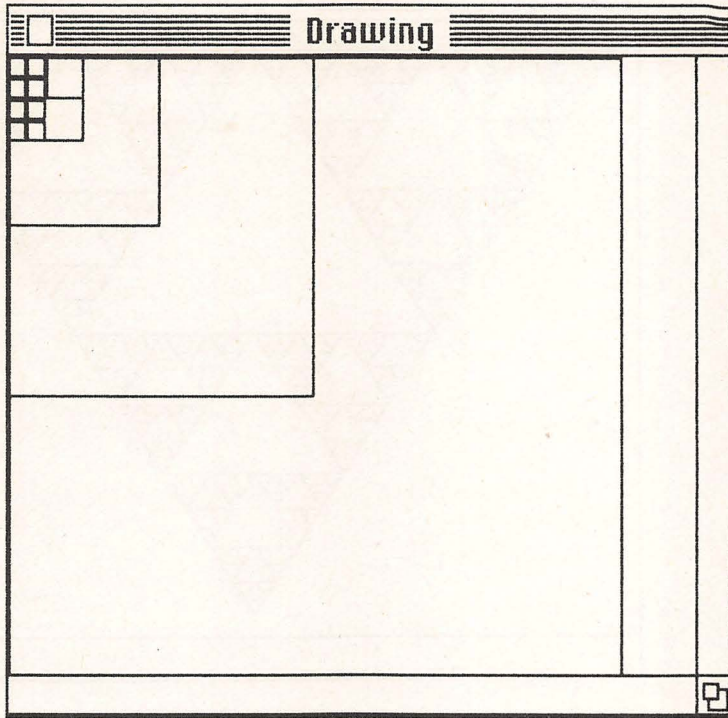


Figure 9-7.

racy. In a procedure such as `LineTo` that requires integer parameters, the `Round` function is used to convert from real to integer. Run this program and observe the order in which the triangles are drawn, which is from smallest to largest (Figure 9-8). This is because the triangles are drawn after the procedure calls rather than before as in the preceding programs.

## Records

In the previous chapter, we were introduced to the difference between scalar types such as `Real`, `Integer`, and `Boolean` and structured types such as `Arrays`. Structured types consist of more than one scalar variable in an organized arrangement. `Arrays` consist of variables of the same data type all referred to by one array name but different subscripts. A `record` is a collection of variables that can be of different data types and are logically

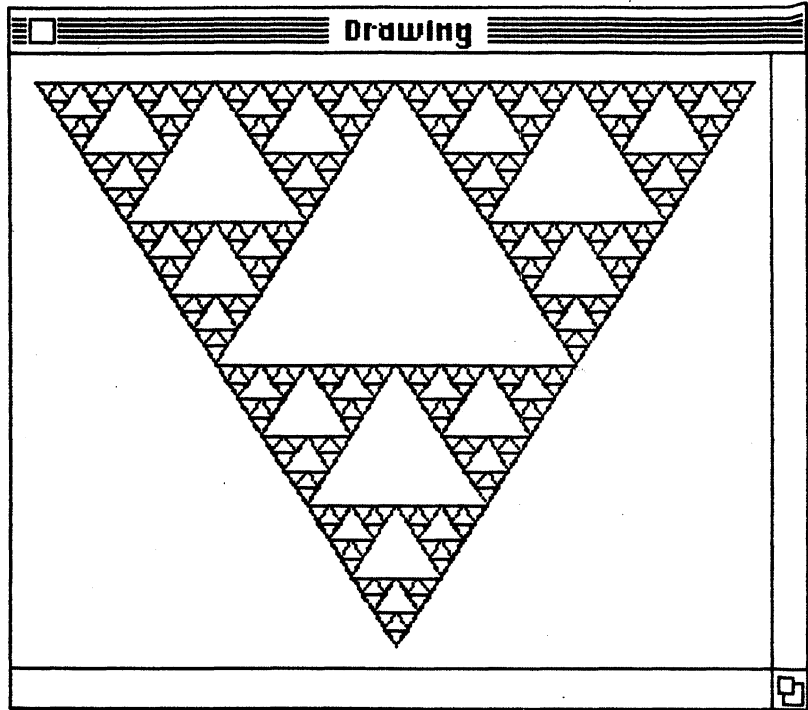


Figure 9-8.

related to each other. For example, all the information about a person—name, age, sex, etc.

A record declaration includes a name of the record and a name and type of each record element.

```
type  
  PayRec=record  
    ID : Integer;  
    Hours : Real;  
    Rate : Real;  
    Pay : Real  
  end;{ PayRec}
```

Examining the declaration we see that PayRec is declared in the Type section. The record type PayRec consists of 4 components, ID of type Integer, and Hours, Rate, and Pay of type Real.

We can now declare a variable to be of type PayRec.

```
var  
  Payroll : PayRec;
```

Here is the syntax diagram for a record declaration.

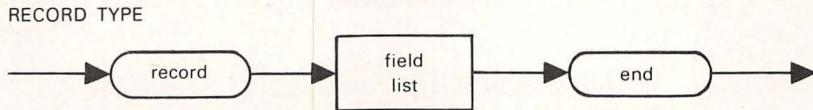


Figure 9-9. Syntax Diagram : **record** Type.

Payroll is now of type PayRec and has 4 components (also called elements or fields). They are referred to as :

Payroll.ID  
 Payroll.Hours  
 Payroll.Rate  
 Payroll.Pay

Each field (or element) is referred to by both its record name and its field name, a point or period is used to separate them. A record element can do anything that any other variable of that type can do. The only difference between a record element and any other variable of that type is that the record element is part of a larger structure. Examples of assignment statements are :

Payroll.ID := 99;  
 Payroll.Hours := 40.0;  
 Payroll.Rate := 12.5;  
 Payroll.Pay := Payroll.Hours \* Payroll.Rate;

If more than one record is declared of the same record type such as:

**var**  
 DayPay, WeekPay : PayRec;

then an entire record can be assigned to another with a simple assignment statement.

WeekPay := DayPay;

Any operation other than simple assignment has to be performed with each separate element. For instance, to multiply all the elements of our record by 5.

WeekPay.ID := DayPay.ID \* 5;  
 WeekPay.Hours := DayPay.Hours \* 5;  
 WeekPay.Rate := DayPay.Rate \* 5;  
 WeekPay.Pay := DayPay.Pay \* 5;



Information is written from a record to the text window or read from the keyboard into a record by using the complete field name.

```
WriteIn(PayRoll.ID);
ReadIn(WeekPay.Rate);
```

This is only a beginning to the use of records. When they are used in conjunction with files they provide a powerful tool for business applications and a way to save data for future use. When used with pointers they can represent a myriad of situations. Both files and pointers will be presented in later chapters.

## The With Statement

When using records it can become tedious to have to use the complete name of a record element over and over. The **with** statement is used to shorten the name of a record element when used in a statement.

```
with Payroll do
  ReadIn(ID);
```

This is the equivalent of:

```
ReadIn(Payroll.ID);
```

The **with** statement automatically prefixes the field name *Id* with the record name *Payroll* to result in *Payroll.Id*.

Examine the syntax of the **with** statement:

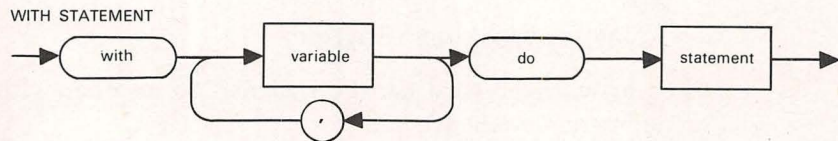


Figure 9-10. Syntax Diagram : **with** Statement

You can see from the diagram that only a single statement is included in it. The power of the **with** statement can be expanded by including in it a compound statement.

```
with Payroll do
  begin
    ID := 10;
```

```

    Readln(Hours,Rate);
    Pay := Hours * Rate
end;

```

This **with** statement replaces :

```

Payroll.ID := 10;
Readln(Payroll.Hours,Payroll.Rate);
Payroll.Pay := Payroll.Hours * Payroll.Rate;

```

## Duplicate Field Names

Because the name of an element of a record is composed of both the record name and the field name, the following variable declarations are possible.

```

var
  Date : 1..31
  Birthdays : record
    Date : 1..31;
    Month : 1..12;
    Year : 1900..1985;
    Age : 1..85
  end;
  Anniversarys : record
    Date : 1..31;
    Month : 1..12;
    Year : 1900..1985;
    Length : 1..85
  end;

```

There are now three variables named **Date**. One is the scalar **Date**, one is in the record **Birthdays** and its complete name is **Birthdays.Date**. The third is in the record **Anniversarys** and its complete name is **Anniversarys.Date**. So actually all three have different names. However, this type of situation could be ambiguous when using a **with** statement. For instance :

```

Date := BirthDays.Date

```

is not the same as:

```

with Birthdays do
  Date:=Date:

```

This last example inside the **with** statement is equivalent to:

```
Birthdays.Date := Birthdays.Date;
```

because the record name that is prefixed to this variable (**Date**) that is a field in the record (**Birthdays**) is assumed for any variable name that is inside the record specified in the **with** statement.

Note that when the records **Birthdays** and **Anniversarys** were declared, they were declared in the **var** section rather than the **type** section. This is permissible and is the same as in declaring any type (see Chapter 6).

More than one record name can be included in the **with** statement.

```
with Birthdays, Anniversarys do
```

```
  Age := Length;
```

is equivalent to:

```
Birthdays.Age := Anniversarys.Length;
```

The proper record name is found and is added to the field name. The position of the record name in the **with** statement is insignificant. It follows then that the **with** statement might have read:

```
with Anniversarys, Birthdays do
```

```
  Age := Length;
```

Using either of the two **with** statements this assignment statement would be illegal:

```
Date := Date;
```

This is because the statement is ambiguous, and it is impossible to tell which field is in which record.

## Arrays Of Records

We can declare an array whose elements consist of records. This is a powerful structure capable of holding a large amount of information easily. Let's design a program that keeps track of the weather over a period of a year.

```
type
```

```
  SkyType = (Sun, Rain, Cloudy);
```

```
  WeatherRec = record
```



```

    Temp : Integer;
    Sky : SkyType
end;
var
    Weather : array [1..365] of WeatherRec;

```

We have set up a type, `WeatherRec`, as a record with two fields. An array was then declared, `Weather`, which has 365 elements (one for each day of the year) of type `WeatherRec`. Thus there are 365 records in the array. When we have an array of records we specify a particular field in a particular record as :

```
Weather[Index].Temp;
```

The record name with its position in the array given as in a subscript is followed by the field name. This array can be pictured like this (Figure 9-11):

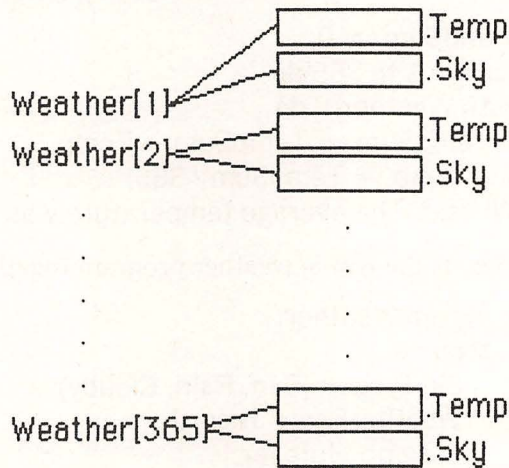


Figure 9-11.

Since each record is an element in the array we always use the array name with a subscript, even when using the `with` statement.

Here the `Temp` field of the first record is read:

```

with Weather[1] do
    Readln(Temp);

```

Here the `Temp` field of the  $I^{\text{th}}$  record is read

```

with Weather[I] do
    Readln(Temp);

```

To continue our weather tracking program let's read the weather for each day of last year.

```
for I := 1 to 365 do
  with Weather[I] do
    begin
      Writeln('Enter temperature for day', I);
      Readln(Temp);
      Writeln('Sun, Rain or Clouds');
      Readln(Sky)
    end; {With}
```

Because of the **with** statement, when Temp was used in the Readln statement it referred to Weather[I].Temp. Note that we can read information into a user-defined type by entering the actual value. This is a MacPascal extension to standard Pascal. In standard Pascal you would have to read the Ord of the User-defined value.

We can now average the daily temperatures.

```
TempSum := 0;
for I:= 1 to 365 do
  with Weather[I] do
    TempSum := TempSum + Temp;
  AvgTemp := TempSum / 365;
  Writeln('The average temperature was', AvgTemp : 6 : 2);
```

Here is the whole weather program together.

```
program Weather;
type
  SkyType = (Sun, Rain, Cloudy);
  WeatherRec = record
    Temp : Integer;
    Sky : SkyType
  end;
var
  Weather : array [1..365] of WeatherRec;
  I, TempSum : Integer;
  AvgTemp : Real;
begin {program}
  for I := 1 to 365 do
    with Weather[I] do
      begin
        Writeln('Enter temperature for day', I : 2);
        Readln(Temp);
        Writeln('Sun,Rain or Clouds');
```

```

        Readln(Sky)
    end; {With}
    TempSum := 0;
    for l:= 1 to 365 do
        with Weather[l] do
            TempSum := TempSum + Temp;
            AvgTemp := TempSum / 365;
            Writeln('The average temperature was', AvgTemp :
                6 : 2)
        end. {program}
    
```

The obvious limitation of this program is that the data entered into the array will be lost as soon as we stop using the program. What is needed is a mechanism to save the data for further use on an external device such as the built-in disk drive. This is the role played by external files and will be covered in Chapter 11.

## Nested Records

The data type of a record can also be a record. This nests one record inside another. Let's examine the following type declarations.

```

type
    FileRec = record
        File1 : string[20];
        File2 : string[20];
        File3 : string[20]
    end;
    DiskRec = record
        DiskName : string[20];
        Contents : FileRec
    end;
var
    Disk1 : DiskRec;
    
```

We now have a record (Disk1), which has as an element a sub-record. Pictorially it looks like Figure 9-12.

The way we assign data to a field in this record will differ depending upon which level of record the field is in. The field DiskName is in the first level and so we need to use the record and the field name:

```
Disk1.DiskName := 'PaulStuff';
```



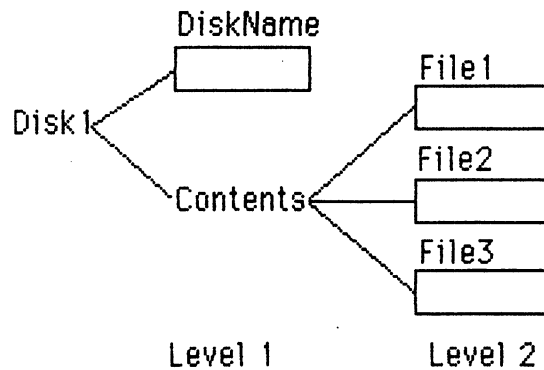


Figure 9-12.

The fields File1, File2, and File3 are all in the second level of the record and their names are composed of the record name, the sub-record name, and the field name:

```
Disk1.Contents.File1 := 'Resume';
```

A field in a record can also be an array. Let's change the declaration of FileRec to:

```
FileRec = record;
  Files : array[1..10] of string[20];
end;
```

The record Disk1 can now be pictured as Figure 9-13.

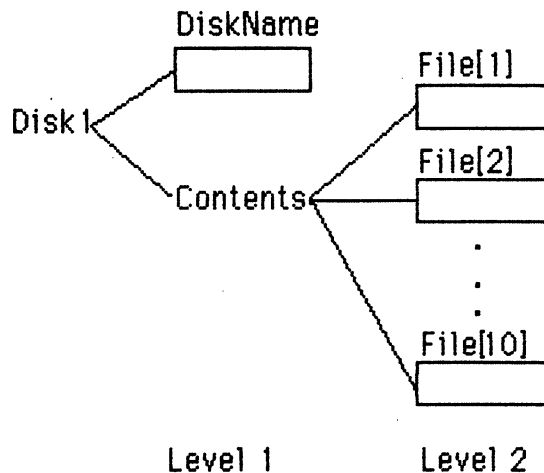


Figure 9-13.

The fields which are array elements are referred to as

Disk1.Contents.Files[I];

Notice that here the subscript is after the field name rather than after the record name as it is when we have an array of records.

## Time And Date Operations

The Macintosh Toolbox contains two procedures to fetch and alter the real time clock built into the computer. To access the clock, MacPascal has a built-in record type defined as:

```
DateTimeRec = record
    Year,
    Month,
    Day,
    Hour,
    Minute,
    Second,
    DayOfWeek : Integer
end;
```

The meaning of most of the field is obvious. Hour is the number of hours since midnight, sometimes referred to as the 24 hour clock. Month is the number of the month from 1 to 12. Day of the week is 1 to 7, from Sunday to Saturday.

To fetch clock information first declare a record of type DateTimeRec.

```
var
    Clock : DateTimeRec;
```

Then read the clock with the GetTime procedure;

```
GetTime(Clock);
```

The fields of Clock now contain the current time and date information. Since all the information is held as integers, we could use User-defined types to display the name of the month and Day.

```
type
    Days = (Sun, Mon, Tue, Wed, Thur, Fri, Sat);
var
    DayName : Days;
```

```
Clock : DateTimeRec;  
I : Integer;  
:  
:  
GetTime(Clock);  
DayName := Sun;  
  for I := 1 to Ord(Clock.DayOfWeek) - 1 do  
    DayName := Succ(DayName);  
  Writeln(DayName);
```

The variable DayName is of type Days which is defined as the days of the week. DayName is initialized to Sun and then a for loop takes the successor of DayName Ord(Clock.DayOfWeek)- 1 time. If it is Sunday, then the statement in the for loop is never executed and DayName stays as Sunday.

## Sets

Sets are a structured data type unique to Pascal among the more popular programming languages. A set is an unordered collection of items of the same data type, called members. Unlike any other type in Pascal, the number of elements in a set may change dynamically. A set is indicated by enumerating members of the set inside of brackets.

|                       |   |
|-----------------------|---|
| [1,3,5,9]             | is the set of odd numbers from 1 to 10. |
| ['A','E','T','O','U'] | is the set of upper case vowels.        |

A subrange can also be used to enumerate the members of a set.

|            |                                       |
|------------|---------------------------------------|
| ['a'..'z'] | is the set of all lower case letters. |
|------------|---------------------------------------|

The general form for the declaration of a set is:

```
var  
  SetName : set of data type;
```

The data type is an ordinal type other than Real. A User-defined ordinal type can also be used.

This declaration creates a set whose members can be upper case letters.

```
var  
  Vowels : set of 'A'..'Z';
```



The declaration of a set does not place any members into it anymore than declaring a variable to be an Integer gives it a value. Members have to be assigned to the set. The members of a set are represented as shown before.

```
Letters := ['A', 'B', 'C'];
```

The set Letters now has three members. The order of the members of a set has no significance. Nor can a set have more than one of the same member. A set with no members is called the empty set. The empty set is represented with two brackets next to each other [].

## Set Operators

To perform operations on sets of the same type there are several set operators.

- + Set Union or addition
- − Set Difference
- \* Set Intersection

### Set Union

Set union forms a third set made up of each of the elements in two sets. Any member appearing in both sets is only included once.

| Expression                  | Result                  |
|-----------------------------|-------------------------|
| $[1,2,3] + [3,4]$           | $[1,2,3,4]$             |
| $['A','C','E'] + ['B','D']$ | $['A','B','C','D','E']$ |

### Set Difference

Set difference forms a third set with the members of the first set that are not in the second set.

| Expression                           | Result                               |
|--------------------------------------|--------------------------------------|
| $[1,2,3] - [3,4]$                    | $[1,2,]$                             |
| $['A'..'Z'] - ['A','E','I','O','U']$ | the set of all upper case consonants |

## Set Intersection

Set intersection forms a third set with all the members that the first and second sets have in common.

| Expression                              | Result      |
|---|-------------|
| $[1,2,3] * [2,4,6]$                     | $[2]$       |
| $['A','b','c','D'] * ['a','b','c','d']$ | $['b','c']$ |

Relational operators also can be used with sets, although their meanings change slightly.

| Expression                     | Returns TRUE if  |
|--------------------------------|--|
| $\text{Set1} = \text{Set2}$    | Set1 and Set2 are identical.   |
| $\text{Set1} < > \text{Set2}$  | The intersection of Set1 and Set2 would produce the empty set.   |
| $\text{Set1} \leq \text{Set2}$ | Set1 is a subset of Set2 (all the members of Set1 are in Set2).  |
| $\text{Set1} < \text{Set2}$    | All the members of Set1 are in Set2 and at least one member of Set2 is not in Set1. Set1 is a strict subset of Set2. |
| $\text{Set1} \geq \text{Set2}$ | Set2 is a subset of Set1 (all the members of Set2 are in Set1).  |
| $\text{Set1} > \text{Set2}$    | All the members of Set2 are in Set1 and at least one member of Set1 is not in Set2. Set2 is a strict subset of Set1. |
| $\text{member in Set1}$        | member is in Set1  |

All of these operators except in work with two sets.

Sets are useful for input verification. If we were writing a program that accepted students' grades of A, B, C, D, and F. We might use the following code.

```
var
  GradeSet : set of 'A'..'Z';
  Grade : 'A'..'Z';
```

```

      .
      .
      GradeSet := ['A','B','C','D','F'];
    repeat
      Writeln('Enter Grade');
      Readln(Grade);
      if Grade not in GradeSet then
        Writeln('Re-enter grade');
    until Grade in GradeSet;

```

Some programs such as the Code Breaker program in Chapter 8 would work more efficiently if it only has to work with upper case characters. For instance, in the Code Breaker program, 26 array elements could be eliminated. To convert from upper to lower case, note that the ASCII codes for the lower case letters are 32 less than the upper case. For example:

'A' equals CHR(ORD('a')+32)

Sets can be used in a program that changes the case of a character.

```

program Convert;
type
  CharSet = set of char;
var
  InString : string[80];
  Ct : Integer;
  LowerCase : CharSet;
  Ch : Char;
begin
  LowerCase := ['a'..'z'];
  Writeln('Enter a lower case string');
  Readln(InString);
  for Ct := 1 to Length(InString) do
    if InString[Ct] in LowerCase then
      InString[Ct] := CHR(ORD(InString[Ct]) - 32);
      {Converting cases}
  Writeln(InString)
end.

```

In program Convert, a set, LowerCase, contains all the lower-case characters. A string is read and a for loop is used to see if any of the characters in the string are a member of LowerCase. If



they are, the character's case is converted and re-assigned to the same position in the **string**.

A similar program can use sets to list all the characters that appear in a **string**. The string is entered and stored in a **string** variable. The individual characters in the string are examined and added to the set. The characters which are members of the set are then displayed. Note that both upper and lower case characters can both be set members.

```
program ExamineCharacters;  
  type  
    CharSet = set of Char;  
  var  
    InString : string[80];  
    LetterSet : CharSet;  
    Ct : Integer;  
    Ch : Char;  
  begin  
    Writeln('Enter a string');  
    Readln(Instring);  
    for Ct := 1 to Length(InString) do  
      LetterSet := LetterSet + [InString[Ct]]; {Add new  
        member to the set}  
    Writeln('These are the characters');  
    for Ch := 'A' to 'z' do  
      if Ch in LetterSet then  
        Writeln(Ch);  
  end.
```

Sets are a powerful Pascal structure and can be used to replace many **if** and **case** statements. Their use helps create elegant, well written programs.

## Exercises

1. What is printed by the following **repeat** loops?
  - a. **M := 5;**  
    **repeat**  
      Writeln(M);  
      **M := M + 3;**  
    **until M > 7;**

```

b. K := 7;
   repeat
     K := K - 1;
     Writeln(K)
   until K = 3;

```

2. Convert the following **repeat** loops into **while** loops.

```

a. Readln(K);
   while K < > -99 do
   begin
     Writeln(K);
     Readln(K)
   end;

```

```

b. P := 5;
   while P < 10 do
   begin
     Writeln(P);
     P := P + 2
   end;

```

```

c. Ch := 'A';
   while Ch < > 'E' do
   begin
     Ch := Succ(Ch);
     Write(Ch)
   end;

```

3. What is printed by the following program segment?

```

var
  A : Integer;
function Fun1(C, D : Real) : Real;
begin
  C := Sqr(C-D);
  Fun1 := C-D
end;
begin {Main program}
  A := 12;
  Writeln(Fun1(A, 3));

```

4. What is printed by the following program?

```

program QuestionFour
var
  A : Integer;

```

```
function Fun2(C : Integer) : Boolean;
begin
  if Odd(C) then
    Fun2 := True;
  else
    Fun2 := False;
end;
function Fun3(C : Integer, D : Boolean) : Integer;
begin
  if D then
    C := C + 1
  else
    C := C * 2;
  Fun3 := C
end;
begin
  A := 17;
  Writeln(Fun3(A, Fun2(A)));
end.
```

5. Write a function that returns the largest of the three parameters it accepts.
6. Write a function that returns the hypotenuse of a right triangle when provided with the other two sides.
7. Define a record that represents the information held in a mailing list.
8. Write a program that uses the record declared in Exercise 7 to accept mailing list information and then prints it.
9. Add a procedure to the weather tracking program developed in the chapter that finds the average daily temperature for the year.
10. Diagram the following record showing the levels of variables as in the style used in the chapter.

```
Amount = record
  Total : Real;
  Minimum : Real
end;
Bills = record
  Payee : string[20];
```



Due : Amount  
end;

11. What are the results of the following set operations?
  - a.  $[1, 3, 5, 7] + [3, 6, 9]$
  - b.  $[1, 3, 5, 7] * [2, 4, 6, 8]$
  - c.  $[1, 3, 5, 7] - [2, 4, 6, 8]$
  - d.  $[1, 2, 3, 4] - [3, 4]$
  - e.  $['A'..'Z'] * ['A', 'E', 'I', 'O', 'U']$
12. What are the results of the following logical operations?
  - a.  $[27] > [26]$
  - b.  $['a', 'b'] = ['b', 'c']$
  - c.  $4 \text{ in } ([1, 2] + [3, 4])$
  - d.  $[1, 2, 3] \geq [1, 2, 3]$
13. Write a program that reads a string and then counts all the vowels in the string.

# 10 A Formal Look at Graphics

In the preceding chapters we have seen “commands” that allowed us to display graphics in the **Drawing** window. We also saw many interesting effects that can be created. These “commands”, which are not part of standard Pascal, are built into the ROM inside the Macintosh. They are actually a library of pre-defined procedures, functions, and data structures which can be accessed as though they are declared in your program. This library, also known as a Unit, is called QuickDraw. QuickDraw is capable of displaying a variety of shapes and text characters in the **Drawing** window. QuickDraw lives up to its name as you have seen from the speed in the graphics program we have written. Actually, QuickDraw is divided into 2 Units, QuickDraw1 and QuickDraw2. All the commands we have employed are contained in QuickDraw1. Normally, you inform Pascal that you want to use a Unit with the **uses** statement.

**uses**

QuickDraw1;

However, since the features of QuickDraw1 are used quite often, this statement is assumed by default. Let's now take a more formal look at some of the components of QuickDraw that we have already seen.

## Points

The most basic QuickDraw data type is the Point. A point can be thought of as the location where two lines on the coordinate

grid intersect. A point is specified by two integers representing the X (vertical) and Y (horizontal) coordinates that intersect. QuickDraw represents a Point in a record which is defined as follows.

```
type
  Point = record of
    V : integer;
    H : integer;
  end;
```

A variable of the type Point can be used to reference a point on the screen.

## Drawing Lines

In Chapter 7 we saw how to draw lines in the Drawing window. Let's take a second look at the procedures used. Lines like all the other objects are drawn with the pen. In order to draw a line we must first position the pen. There are two procedures used to do this, one which we have already seen.

```
procedure MoveTo(X, Y : Integer);
```

The MoveTo procedure positions the pen at the point specified by the horizontal coordinate X and the vertical coordinate Y. Notice how the procedure was specified. In this chapter, the QuickDraw procedures will be shown as they are declared in QuickDraw. This will give you an opportunity to examine the data types of the parameters used and whether they are variable or value parameters.

The second procedure which moves the pen relative to its old position is Move.

```
procedure Move(X, Y : Integer);
```

X and Y are added to the current horizontal and vertical positions of the pen, respectively, to yield the new current position. For example, let's examine the position of the pen after each of the following two statements are executed.

```
Moveto(3, 2);
Move(4, 2);
```

After the Moveto statement, the pen is located at point (3, 2).



After the Move statement the pen is moved to point (7, 4) (can be thought of as point (3+4, 2+2)).

There are two procedures that work in exactly the same manner as Move and MoveTo except that they draw a line between the old pen position and the new pen position.

```
procedure Line(X, Y : Integer);
procedure LineTo(X, Y : Integer);
```

The following program illustrates the use of the pen positioning and line drawing procedures (Figure 10-1).

```
program test;
uses
    QuickDraw1;
var
    TempRect : Rect;
    I, J : Integer;
begin
    I := 20;
    J := 491;
    while I <= 491 do
        begin
            MoveTo(I, Round(100 * Sin(I / 120)) + 150);
            LineTo(J, -Round(100 * Sin( J / 120)) + 160);
            I := I + 5;
            J := J - 5;
        end
    end.
```

## Rectangle

The first graphics data structure we saw in Chapter 4 was the Rectangle. In that chapter, we saw variables declared of the data type Rect. Rect is actually defined as a record type, let's look at its declaration.

```
type
    Rect = record of
        Top : Integer;
        Left : Integer;
        Bottom : Integer;
        Right : Integer
    end;
```

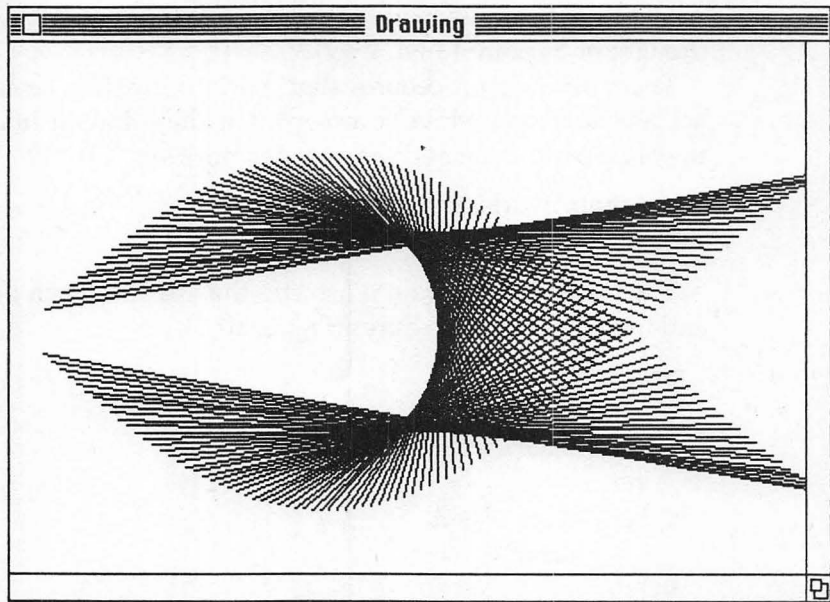
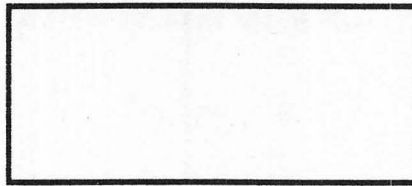


Figure 10-1.

The fields in Rect define lines in the coordinate grid which enclose the rectangle.

(Upper, Left)



(Lower, Right)

Figure 10-2.

Top and Left are the Y and X coordinate of the upper left hand point in the rectangle and Bottom and Right are the Y and X coordinates of the lower right hand point.

Values can be assigned to a variable of type Rect as in any record.

```
var
  TempRect : Rect;
  :
```

```

:
TempRect.Top := 10;
TempRect.Left := 20;
TempRect.Bottom := 140;
TempRect.Right := 78;

```

In Chapter 4, we didn't do this. Instead we used the `SetRect` procedure which does the assignments more conveniently. Remember that defining a rectangle does not display it in the window, a routine must be called to do that.

The definition of `SetRect` is :

```

procedure SetRect(var R : Rect;
Left, Top, Right, Bottom : Integer);

```

Notice that in the procedure declaration of `SetRect`, the lines are in the order Left, Top, Right, and Bottom, which is in the same order as in the `Rect` type declaration.

The four assignment statements used before values to assign values to the rectangle could be accomplished with the single statement:

```

SetRect(TempRect,20,10,78,140);

```

## Controlling the Drawing Window

The size of the **Drawing** window can be controlled by your program. Two procedures are used to accomplish this.

```

procedure SetDrawingRect(R : Rect);
procedure ShowDrawing;

```

`SetDrawingRect` sets the size of the **Drawing** window to that specified by the rectangle `R`.

`ShowDrawing` displays the **Drawing** window on the Macintosh's screen. For example, to make the **Drawing** window as big as the entire screen the following statements could be used:

```

SetRect(TempRect, 0, 0, 511, 341);
ShowDrawing(TempRect);

```

We have already seen the analogous procedures for setting the size of the **Text** window on the screen. These procedures are:

```

SetTextRect(R : Rect);
ShowText;

```



## Drawing Rectangles

We have already learned how to draw and erase rectangles with the `FrameRect` and `EraseRect` procedures. There are a total of five procedures for drawing rectangles.

**procedure** `FrameRect`(`R : Rect`);

`Frame` draws a box that is enclosed by the rectangle specified by `R`.

**procedure** `PaintRect`(`R : Rect`);

`PaintRect` fills the rectangle specified by `R` with black.

**procedure** `EraseRect`(`R : Rect`);

`EraseRect` erases the rectangles indicated by `R`.

**procedure** `InvertRect`(`R : Rect`)

`InvertRect` inverts (if the pixel was white make it black, if the pixel was black make it white) the area enclosed by the rectangle specified by `R`.

**procedure** `FillRect`(`R : Rect`, `Pat : Pattern`);

`FillRect` draws a rectangle filled with the pattern indicated by the second parameter. The valid patterns are:

White  
Black  
Gray  
LtGray {light gray}  
DkGray {dark gray}

The following program illustrates the use of `FillRect`.

**program** `Fills`;

**var**

`Rect1`, `Rect2` : `Rect`;

**begin**

`SetRect`(`Rect1`, 10,10, 30,40);

`FillRect`(`Rect1`, `Gray`);

`SetRect`(`Rect2`, 65,65,100,105);

`FillRect`(`Rect2`, `DkGray`)

**end.**

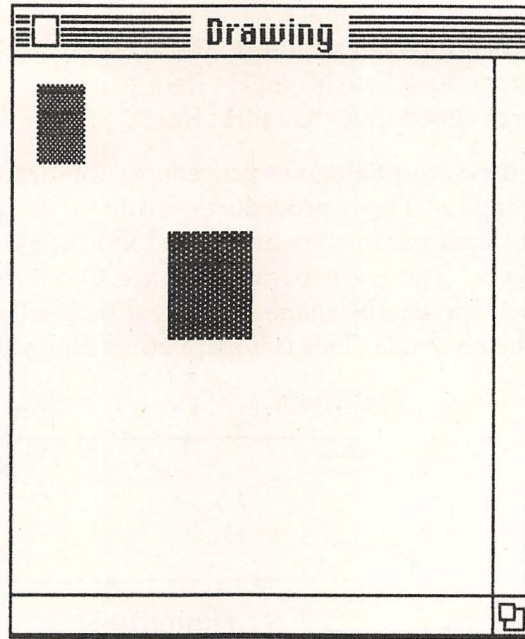
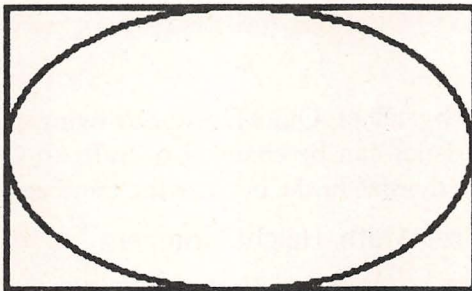


Figure 10-3.

## Other Shapes

Similar procedures exist for drawing ovals and round cornered rectangles. The procedures for ovals work in exactly the same manner as those for rectangles. The size and shape of the oval is determined by inscribing an oval in a rectangle that is passed to the oval drawing procedures. See Figure 10-4.



**Oval inscribed  
in a rectangle**

Figure 10-4.

The oval drawing procedures are:



```

procedure FrameOval(R : Rect);
procedure PaintOval(R : Rect);
procedure EraseOval(R : Rect);
procedure InvertOval(R : Rect);

```

There are analogous procedures for drawing round cornered rectangles. These procedures work in the same fashion except additional parameters are passed indicating the roundness of the corners. The extra parameters are *OvalWidth* and *OvalHeight* which specify the shape of an oval that is "fitted" into the corner of the rectangle. This is illustrated by Figure 10-5.

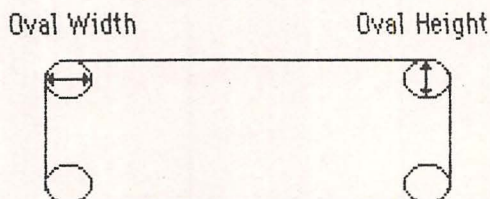


Figure 10-5.

The round corner rectangle procedures are:

```

procedure FrameRoundRect(R : Rect;OvalWidth,OvalHeight
                           : Integer);
procedure PaintRoundRect(R : Rect;OvalWidth, OvalHeight
                           : Integer);
procedure EraseRoundRect(R: Rect;OvalWidth, OvalHeight
                           : Integer);
procedure InvertRoundRect(R : Rect;OvalWidth, OvalHeight
                           : Integer);

```

## The Pen

The pen is used by all of QuickDraw's drawing procedures. The pen's characteristics can be changed by calls to QuickDraw routines. To change the size of the pen use the *PenSize* procedure.

```

procedure PenSize(Width, Height : Integer);

```

Width and Height specify the size of the pen. The default pen size is one pixel high by one pixel wide.

The color or pattern contained in the pen can also be changed using the *PenPat* procedure.



```
procedure PenPat(Pat : Pattern);
```

The color of a pixel drawn on the screen is not necessarily the color of the pixels contained in the pen. The color of a pixel displayed on the screen is determined by three things, the pattern in the pen, the color of the pixel already on the screen, and the mode of the pen. The mode of the pen determines how pixels on the screen and the pattern in the pen interact. The mode of the pen can be altered by using the PenMode procedure.

```
procedure PenMode(Mode : Integer);
```

Valid values for mode are the predeclared QuickDraw constants: PatCopy, PatOr, PatXor, PatBic, NotPatCopy, NotPatOr, NotPatXor and NotPatBic. For example, let us look at the default pen mode PatCopy. If the pen is black, then the pixel on the screen will be black no matter if the pixel was black or white before. If the color in the pen is white then the pixel on the screen will end up white without regard to its previous color. This is graphically represented in Figure 10-6.

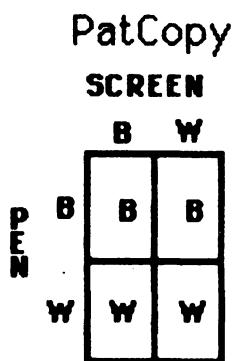


Figure 10-6.

The purpose of the various pen modes is to make it possible to draw on different color backgrounds. The PatCopy pen mode corresponds to a real life pen drawing on paper. The actions of each of the pen modes can be understood by examining the charts in Figure 10-7.

The following program demonstrates several of the different pen modes against different backgrounds.

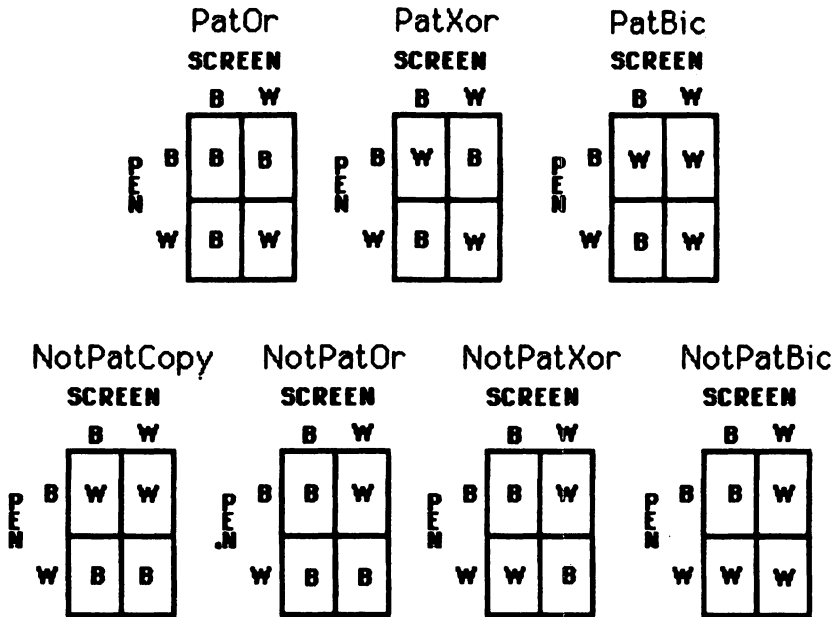


Figure 10-7.

```

program ModeDemo;
var
  DrawingWindow : Rect;
begin
  { set up screen }
  HideAll;
  SetRect(DrawingWindow, 40, 40, 400, 320);
  SetDrawingRect(DrawingWindow);
  ShowDrawing;
  { Draw Color bands }
  FillRect(10, 80, 240, 120, Black);
  FillRect(10, 120, 240, 160, Gray);
  FillRect(10, 160, 240, 200, LtGray);
  FillRect(10, 200, 240, 240, DkGray);
  FillRect(10, 240, 240, 280, White);
  FrameRect(10, 240, 240, 280);
  PenSize(5, 5);
  {Draw first line, default mode}
  PenPat(Black);
  Moveto(60, 20);
  Lineto(300, 20);
  {Draw second line}
  PenMode(PatCopy);

```

```
Moveto(60, 40);
Lineto(300, 40);
{Draw third line}
PenMode(PatOr);
Moveto(60, 60);
Lineto(300, 60);
{Draw fourth line}
PenMode(PatXor);
Moveto(60, 80);
Lineto(300, 80);
{Draw sixth line}
PenMode(PatBic);
Moveto(60, 100);
Lineto(300, 100);
{Draw seventh line}
PenMode(NotPatCopy);
Moveto(60, 120);
Lineto(300, 120);
{Draw eighth line}
PenMode(NotPatOr);
Moveto(60, 140);
Lineto(300, 140);
{Draw ninth line}
PenMode(NotPatXor);
Moveto(60, 160);
Lineto(300, 160);
{Draw tenth line}
PenMode(NotPatBic);
Moveto(60, 180);
Lineto(300, 180);
end.
```

The output of the program is shown in Figure 10-8.

To get a real feel for the purpose of each pen mode do some experimentation with each by modifying the ShowMode program.

## The Mouse

Procedures that are not part of the QuickDraw unit, but that are very useful in conjunction with graphics are the ToolBox's mouse routines. When the mouse is moved, the position of the



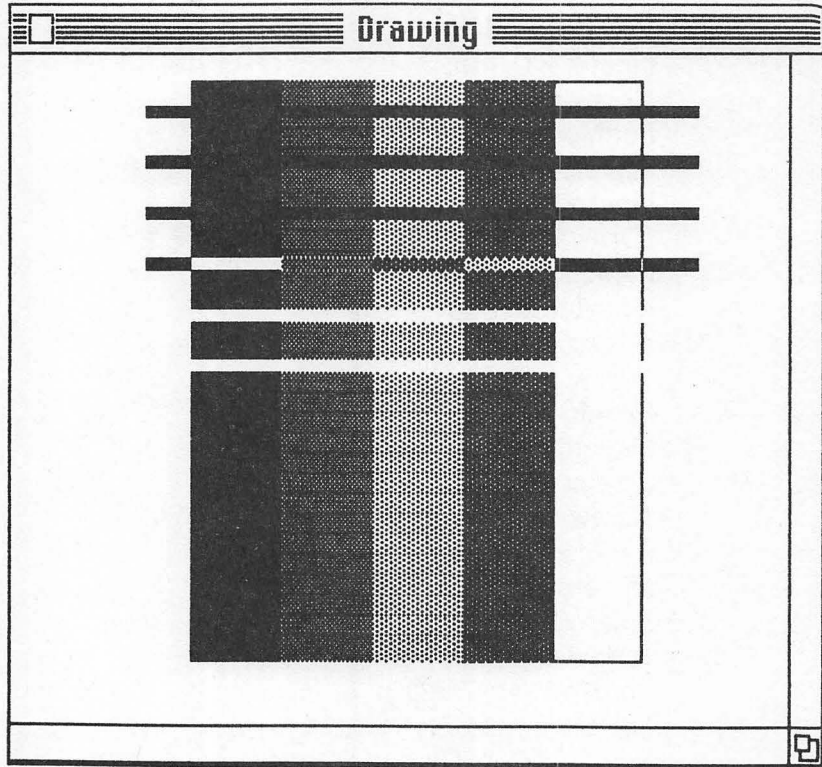


Figure 10-8.

cursor on the screen moves. MacPascal has a procedure that returns the coordinates of the cursor on the screen.

```
procedure GetMouse(var X, Y : Integer);
```

X and Y are variables that contain the horizontal and vertical coordinates, respectively, of the cursor on the screen. The MouseDemo program demonstrates the use of GetMouse. Run the program and move the cursor around the screen using the mouse. The coordinates of the cursor will be displayed in the text window. Notice that the coordinates are relative to the upper left corner of the drawing window which has the coordinate (0,0). To exit this program use the pause option on the menubar.

```
program MouseDemo;
var
    Horizontal, Vertical : Integer;
begin
    while true do
```

```

begin
  GetMouse(Horizontal, Vertical);
  Writeln('Horizontal = ' : 15, Horizontal : 3);
  Writeln('Vertical = ' : 15, Vertical : 3)
end
end.

```

Other mouse routines are functions that return the status of the mouse's button.

```

function Button : Boolean;
function StillDown : Boolean;
function WaitMouseUp : Boolean;

```

The function `Button` returns true if the button is being held down and false if it is not. The function `StillDown` and `WaitMouseUp` are very similar functions. `StillDown` returns true if the button is down and has not been released since the button was originally tested. If the button is released and is pressed again between calls to `button` then `StillDown` returns false. `WaitMouseUp` returns true if the button is down and has not been released since the button was last tested by the `Button` function. `StillDown` and `WaitMouseUp` both return false if the button is not being pressed or was released since the last time the button was tested.

## The Cursor

There are two important `QuickDraw` routines that control whether or not the cursor is displayed on the screen.

```

procedure ShowCursor;
procedure HideCursor;

```

These procedures act exactly as you think they would from their names. `HideCursor` hides the cursor from the screen and `ShowCursor` restores it. It is important to notice that when the cursor is hidden it still exists and is at some location on the screen. The cursor can be moved by the mouse even when it is hidden from view. Verify this by running the `MouseDemo` program again, but with a call to `HideCursor` before the while loop.

## SketchPad

Now that we have some tools to work with, let us write a simple application program that allows us to draw on the Macin-

tosh's screen. Let us first look at a simple English-like pseudocode for our program.

```

initialize drawing window
repeat
  if button still down then
    draw a line from last position to current position
  otherwise if the button was pressed
    make current position the last position
    (starting position of new line)
until forever

```

This pseudocode is easily translated into the following program: (See Figure 10-9)

```

program SketchPad;
{ Make the screen a drawing pad and the mouse a pen }
uses
  QuickDraw1;
const
  Forever = False;
var
  Pnt : Point; { holds the cursor's position on the screen }
  TempRect : Rect;
begin
  { Make the entire screen a sketch pad }
  SetRect(TempRect, 0, 0, 511, 341);
  SetDrawingRect(TempRect);
  ShowDrawing;
repeat
  { using individual components of point data type}
  GetMouse(Pnt.H, Pnt.V);
  if WaitMouseUp then
    LineTo(Pnt.H, Pnt.V)
  else if Button then
    MoveTo(Pnt.H, Pnt.V);
until Forever
end.

```

This program works fine for what it is, a simple sketch program that duplicates the capabilities of a pencil and paper. We could make this program more powerful if we could use all of the features discussed above, interactively, to change the characteristics of our pen and to draw different types of shapes. We would



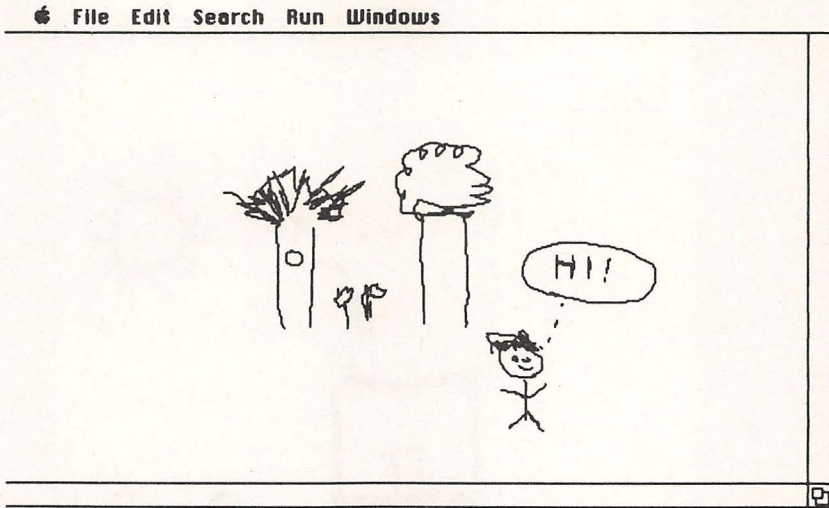


Figure 10-9.

like to make these features easy to use and consistent with the Macintosh user interface.

## A New Feature

The new feature we will add is to change the size of the pen used to draw on the screen. We will implement this by creating two boxes on the screen. One box will have the word "Bigger" in it and the other will have the words "Smaller" in it (Figure 10-10). When the cursor is placed inside the "Bigger" box and the button is clicked the size of the pen will increase. When the cursor is put into the "Smaller" box the size of the pen will decrease. In order to implement this design there are several more QuickDraw routines we must learn about.

## Displaying Text on the Drawing Window

There are two important procedures for putting text in the drawing window.

**procedure DrawString(S : string);**

The DrawString procedure displays the string constant or variable specified by S onto the drawing window at the current pen location. The upper left corner of the first character in the string

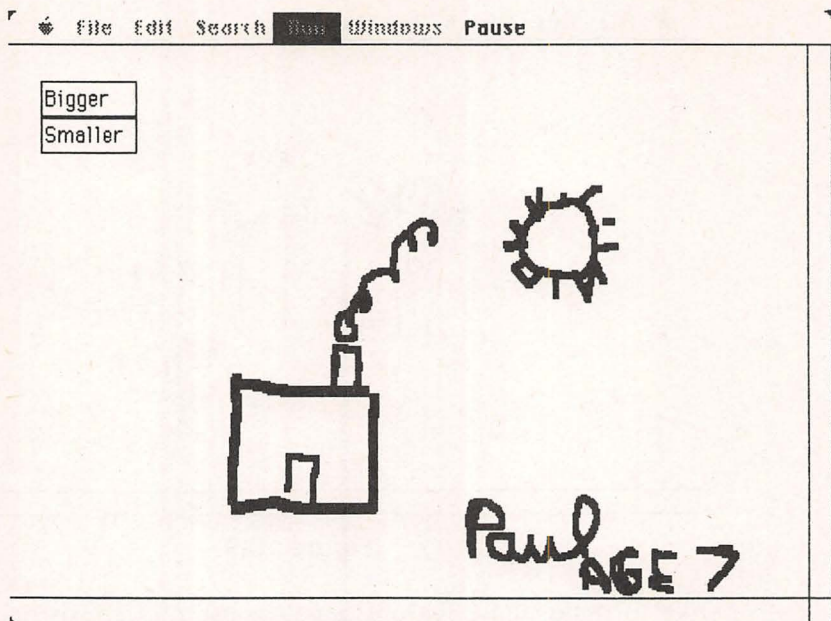


Figure 10-10. SketchPad with new feature.

is placed exactly on the current pen location and all subsequent characters are placed to the right.

**procedure** DrawChar(Ch : Char);

The DrawChar procedure works exactly the same way except it only writes a single character on the screen.

Several other QuickDraw routines allow you to change the attributes of the characters displayed on the screen.

**procedure** TextFont(Font : Integer);

**procedure** TextSize(Size : Integer);

**procedure** TextFace(Face : Style);

TextFont determines the shape of the letters that will be displayed. The default value for font is zero and will display the system font (Chicago). If no font for a value of font is available the default font will be displayed. Experiment with other values for font to see what fonts are available on your machine. Some of the fonts that might be available are shown in Figure 10-11.

TextSize determines the size in points of the character being displayed. The default for size is 0 which picks the system font size of 12. Examples of different font sizes are shown in Figure 10-12.

| <u>Font</u>     | <u>Value</u>           |
|-----------------|------------------------|
| <b>Chicago</b>  | <b>0 (system font)</b> |
| <b>New York</b> | <b>2</b>               |
| <b>Geneva</b>   | <b>3</b>               |
| <b>Monaco</b>   | <b>4</b>               |
| <b>Venice</b>   | <b>5</b>               |
| <b>London</b>   | <b>6</b>               |
| <b>Athens</b>   | <b>7</b>               |

Figure 10-11.

**9 points**  
**10 points**  
**12 points**  
**14 points**  
**18 points**

Figure 10-12.

TextFace selects the special characteristics of the characters displayed. The type Style is defined as a set of pre-defined constants.

**type**

StyleItem = (Bold, Italic, Underline, Outline,  
Shadow, Condense, Extend);

Style = **set of** StyleItem;

Example of some of the different characteristics available are:



Normal

**Bold**

*italics*

underline

**shadow**

**outline**

Figure 10-13.

For example, to set the text face to italic you could use the procedure call:

```
TextFace([Italic]);
```

Notice that italic is in brackets because it is a member of a set of type style. To set the text face to both italic and underlined you would use:

```
TextFace([Italic, Underline]);
```

To set the text face back to normal you would use the empty set:

```
TextFace([ ]);
```

## Calculations with Rectangles

For our programs we need to know if a point falls inside a rectangle or if two rectangles are touching each other.

PointInRect returns true if the point Pt is inside the rectangle specified by R.

```
function PointInRect(Pt : Point; R : Rect) : Boolean;
```

SectRect returns true if FirstRect and SecondRect intersect (any part of the two rectangles occupy the same location on the screen). The rectangle variable ThirdRect becomes the rectangle that encloses the intersected area.

```
function SectRect(FirstRect,SecondRect:Rect;  
var ThirdRect:Rect) : Boolean;
```

## Implementing the New Feature

Now that we have the required tools we can implement the new features into our SketchPad. To implement the new features we will add several parts to our old SketchPad program.

1. We must initialize our "Bigger" and "Smaller" box, draw them on the screen and label them. This will be done in the new initialization procedure.
2. We must check each time through the loop if the user moved the mouse into (selected) the "Bigger" or "Smaller" boxes. This will be done with a call to the new function `Select`, which returns true if the cursor was placed in the box and the button was pressed. To be consistent with the rest of the Macintosh environment we will invert the box when the cursor is in it and the button is pressed (the box is selected) and un-invert it when the button is released.
3. If the box was selected we take the appropriate action. In this case it means calling the procedure `ChangePenSize` to increment or decrement the size of the pen.

```

program SketchPad;
  uses
    QuickDraw1;
  const
    Forever = false;
  var
    Pnt : Point;
    Xpen, Ypen : Integer;
    BiggerBox, SmallerBox, TempRect : Rect;
  procedure Initialize;
  begin
    { Prepare Screen for Drawing Graphics }
    SetRect(TempRect, 0, 0, 511, 341);
    SetDrawingRect(TempRect);
    ShowDrawing;
    { Draw Selection Boxes on Screen }
    SetRect(BiggerBox, 20, 40, 80, 60);
    SetRect(SmallerBox, 20, 60, 80, 80);
    FrameRect(BiggerBox);
    FrameRect(SmallerBox);
  end

```

```
    MoveTo(22, 54);
    DrawString('Bigger');
    MoveTo(22, 74);
    DrawString('Smaller');
    { Initialize PenSize }
    Xpen := 1;
    Ypen := 1
end;
function Select (Box : Rect) : Boolean;
{ Returns True if Box was specified }
begin
    Select := False;
    if PtInRect(Pnt, Box) then
        if Button then
            begin
                InvertRect(Box);
                repeat
                    until not Button;
                InvertRect(Box);
                Select := True
            end
        end;
end;
procedure ChangePenSize (I : integer);
begin
    Xpen := Xpen + I;
    Ypen := Ypen + I;
    PenSize(Xpen, Ypen)
end;
begin
    Initialize;
    repeat
        GetMouse(Pnt.H, Pnt.V);
        if Select(BiggerBox) then
            ChangePenSize(1)
        else if Select(SmallerBox) then
            ChangePenSize(-1);
        if WaitMouseUp then
            LineTo(Pnt.H, Pnt.V)
        else if Button then
            MoveTo(Pnt.H, Pnt.V);
    until Forever
end.
```



This same technique can be used to add many other easy-to-use features to this and other programs.

## Playtime with QuickDraw

It is about time we had a bit of fun, so let us make use of our newly acquired skills in programming Pascal to design and write a game. We will use a top down approach that will allow us to develop the overall structure of the game first and develop the details as we go along.

We will write a racket game where the player controls a racket using the mouse and attempts to hit a ball against a wall. If the ball goes by the racket, the ball goes out of play, and the game is over.

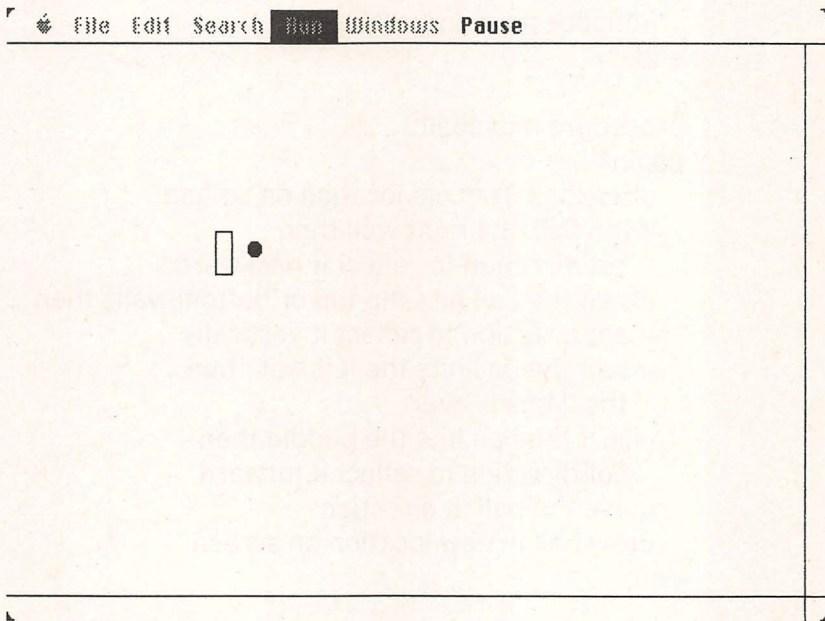


Figure 10-14.

Let us look at a very rough outline of the game.

```
program PaddleBall;
begin
  initialize;
  while not game over do
    begin
```

```
        movepaddle;  
        moveball  
    end  
end;
```

The above program, even though it is in pseudocode can actually be used as our main program with a few additions such as variable declarations. Let us continue to develop our program by developing each of the procedures in the above program.

```
procedure initialize;  
begin  
    initialize score to zero  
    initialize ball shape and position  
    initialize ball direction  
    initialize paddle shape and position  
    initialize boundary walls  
end;
```

```
procedure moveball;  
begin  
    erase ball from old location on screen  
    if the ball hits right wall then  
        set direction to reflect it backward  
    else if the ball hits the top or bottom walls then  
        set direction to reflect it vertically  
    else if the ball hits the left wall then  
        the game is over  
    else if the ball hits the paddle then  
        set direction to reflect it forward  
    move the ball in direction  
    draw ball in new location on screen  
end;
```

```
procedure movepaddle;  
begin  
    erase paddle from screen location  
    get location of mouse controlled cursor  
    draw paddle at new location  
end;
```

We can achieve animation effects on the screen by drawing some object on the screen, leaving it there a short period of time,

and then erasing it and drawing it a short distance away. If this process is done repeatedly at high speed, the drawn object appears to move smoothly across the screen. In the pseudocode, both the paddle and the ball are animated in this way.

Now that the rough design for our game is complete, we can turn our attention to the details of how our graphics objects move and how they can be represented using the Macintosh's Quick-Draw routines.

Let us now examine in more detail the procedure that will move the ball around the screen. There are several factors to consider when moving the ball, the direction the ball is heading, the position of the ball on the screen, and whether or not the ball hits a wall or the paddle.

We will need a variable to keep track of which horizontal direction the ball is traveling.

**var**

Forward : Boolean; { true means ball moves right }  
                          { false means ball moves left }

In order to keep track of how many pixels the ball should travel in each direction on the screen, we will need two variables, DispHoriz to track the number of pixels to move horizontally and DispVert to track the number of dots to move vertically. Since screen coordinates are integer values these variables can be integers.

We also want the ball to travel at angles so we will declare a variable Slope that will hold the angle of the ball's travel. The slope will tell how many pixels the ball travels up or down for every pixel it travels sideways. For example to move due east, the direction would be right and the slope would be 0. No up or down movement just movement to the right.

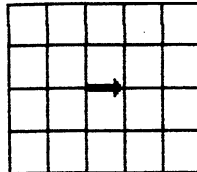


Figure 10-15.

To move the ball north northwest, the direction would be left and the slope would be -2, two pixels up for every one pixel to the left.



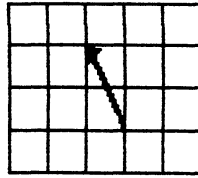


Figure 10-16.

This representation of direction allows us to specify precisely every direction except for straight up and straight down. The slope for these directions would be infinity and negative infinity, respectively. Since the computer cannot represent these numbers the ball will not be able to travel in these directions. This is fine because the ball would never reach the paddle on the left side of the screen if it were to travel vertically. When the ball hits a wall or paddle, it should reflect off the wall in a natural manner.

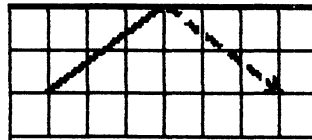


Figure 10-17.

Before the ball hits the top wall it is traveling  $X$  pixels *up* for every pixel it is traveling across. After bouncing off the top wall, the ball is traveling  $X$  pixels *down* for every pixel across. The horizontal direction of the ball does not change. Therefore only the sign of the slope changes when the ball hits the top wall. The same is true for the bottom wall. When the ball bounces off the right wall or the left side of the paddle, the slope remains the same but the direction changes.

```

program PaddleBall;
{ Play a game of hit the ball off the wall but don't let it get
  by }
uses
  QuickDraw1;
var
  Ball, Paddle, Top, Bottom, Left, Right, TempRect : Rect;
  Difficulty, XPaddle, YPaddle, I, J, DispHoriz, DispVert,
    Slope : Integer;
  Gameover, Forward : Boolean;
procedure Init;
```

```

begin
  { Set the screen up }
  SetRect(TempRect, 0, 0, 512, 342);
  SetDrawingRect(TempRect);
  ShowDrawing;
  HideCursor;
  { Size of ball is 9 by 9 }
  SetRect(Ball, 0, 0, 9, 9);
  { Set the boundaries of the game }
  SetRect(Left, 0, 11, 1, 332);
  SetRect(Top, 0, 20, 502, 21);
  SetRect(Bottom, 0, 325, 502, 326);
  SetRect(Right, 498, 10, 511, 332);
  { Initial Position of Ball is 100,100 }
  DispHoriz := 100;
  DispVert := 100;
  { Set initial direction of ball }
  Slope := 2;
  Forward := true;
  { Set initial position of paddle }
  Difficulty := 0;
  Gameover := false
end;
procedure MovePaddle;
{ Erase paddle and redraw at new location }
var
  XMouse, YMouse : Integer;
begin
  GetMouse(XMouse, YMouse);
  EraseRect(Paddle);
  SetRect(Paddle, Difficulty, YMouse, Difficulty + 11,
    YMouse + 25);
  FrameRect(Paddle);
end;
procedure MoveBall;
{ Display ball in appropriate location on screen taking }
{into account reflection of the ball off the borders}
begin
  if SetRect(Right, Ball, TempRect) then
    begin
      SysBeep(1);
      Forward := False
    
```

```

    end
    else if SectRect(Left, Ball, TempRect) then
    { Hit left wall, game over }
      Gameover := true
    else if SectRect(Top, Ball, TempRect)
      or
      SectRect(Bottom, Ball, TempRect) then
    begin
      SysBeep(1);
      Slope := -Slope
    end
    else if SectRect(Paddle, Ball, TempRect) and (not
      Forward) then
    begin
      SysBeep(1);
      Forward := true;
      Difficulty := Difficulty + 10;
    end;
    if Forward then
      DispHoriz := DispHoriz + 10
    else
      DispHoriz := DispHoriz - 10;
      DispVert := DispVert + Slope;
      EraseOval(Ball);
      SetRect(Ball, DispHoriz, DispVert, DispHoriz + 9,
        DispVert + 9);
      PaintOval(Ball)
    end;
    begin {Main program}
      Init;
      while not Gameover do
      begin
        MoveBall;
        MovePaddle;
      end
    end
  end. {Main program}

```

## Exercises

1. Add some of the following features to the SketchPad program:

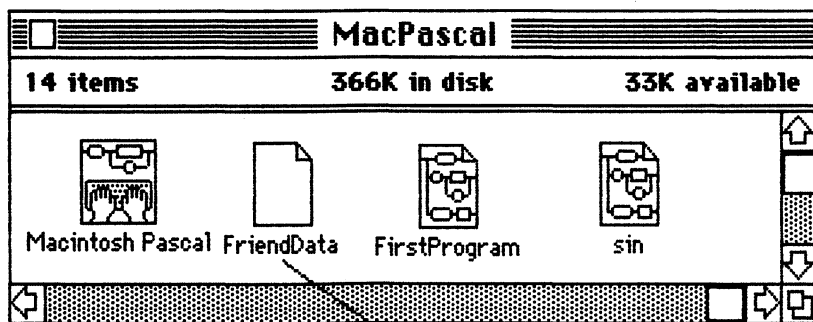


- a. Change color of pen.
  - b. Change penmode.
  - c. Allow selection of different shapes, rectangles, ovals and round cornered rectangles ( Hint: use the animation technique of drawing then erasing the shapes while the mouse's button is down. Draw a final version of the shape when the button is released.
2. Add the following features to the paddleball program
  - a. Add scoring.
  - b. Change shape of boundaries. How would ball reflect off a sloping wall?
  - c. Add obstacles on the playing field.
  - d. Add other animated objects such as another ball or other moving objects.
3. Write a generalized menu procedure that can be used to present a mouse driven menu to the user.

# 11 Files

Computers would not be very useful if they were limited only to information that could fit in their memory. To overcome this limitation computers can store information on secondary storage devices such as disk drives and bring this information into memory only when needed.

Information is stored on a disk in a file. A file is a collection of components all of the same data type. This is similar to an array but there are significant differences between files and arrays. An array is kept in memory and is limited to the number of elements it was declared to have. A file is maintained on a secondary storage device (usually a disk) and has no fixed size, components can be added or deleted at any time. Because a file is kept on disk it is in one sense independent of the program that created it. A file can continue to exist after the program that built it has terminated. A file created by a MacPascal program can be seen in the Finder's Desktop with an icon and its own name.



File on disk

Figure 11-1.

An example of a file that you are already familiar with is a text file (a file whose components are of type `char`). All source codes for MacPascal programs that are entered and saved are stored in text files.

In this chapter, we will be looking at the two ways to store and access data in a file, sequential and random access. In a sequential file, all the components must be accessed in the order in which they were placed into the file. This can be thought of as being similar to a cassette tape, to hear a song recorded at the end of the tape all the songs before it must be passed over. In a random file, any particular component can be accessed in any order. This is sometimes called direct access. This is like moving the tone arm of a phonograph to the particular song you want to hear without playing all the songs before it. Random access is accomplished by numbering each component of a file starting at zero. This is called the record number because traditional data processing terminology labels the components of a file a record. Don't confuse this use of the term record with the record data type. The records in a file may contain data whose data type is a record or any other valid Pascal data type.

A file is created in the variable declaration section of a program or procedure by declaring a file name and the type of the file's components.

```

type
  PersonInfo = record
    FirstName,
    LastName : string[20]
    Phone : string[10]
  end; { PersonInfo }
var
  People : file of PersonInfo;
  Numbers : file of integer;

```

The above `var` section creates two files, `People` and `Numbers`. The following syntax diagram describes the syntax of a file declaration.

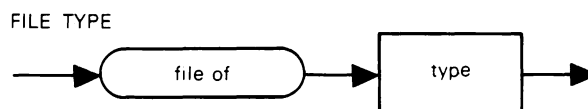


Figure 11-2. Syntax Diagram : File type



The data type of the components of a file can be any standard Pascal type or any type created in a **type** declaration. Very often the components of a file are declared as a record. In *People*, the components will be of the type *PersonInfo*. All the components of *Numbers* will be integers. A file variable such as *People* and *Numbers* are different from other types of variables in that they can be used only with the file procedures which we will soon see. No other operations such as assignment can be performed with them.

## The File Buffer

A file is accessed through a special type of variable called a file buffer. A file buffer can be thought of as a window into the file which points to a particular file component. Through the file buffer, information can actually be placed in or retrieved from the disk. When the window is pointing to a record in the file, that record can be retrieved or altered. For every file that is declared, a file buffer variable is automatically created. It is through this variable that information will actually pass through. The file buffer variable is accessed by using the file name followed by an up arrow  $\wedge$  (typed on the Mac as a Shift-6). If *Numbers* is declared to be a file of integers, then the file buffer associated with that file is *Numbers* $\wedge$ . A file buffer can be used like any other variable of the same data type. For example:

```
with People $\wedge$  do  
  ReadIn(FirstName, LastName, Phone);
```

## Using Files

There are three major steps in using files: opening the file, accessing the file, and closing the file. For each of these steps Pascal provides built-in procedures for accomplishing each of these tasks.

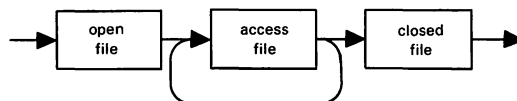


Figure 11-3.

## Opening Files

Associated with each file we use are two names. The first is the file variable declared in a program which is used in the program to refer to the file. This is sometimes referred to as the logical file name. The other name is the name that the file will actually be called on the disk. This is called the physical file name. It is the one you can see next to the icon for the file in the Desktop. The logical and physical file names are linked together when you open the file in a program.

There are three procedures provided by MacPascal for opening files: Rewrite, Reset, and as we shall see later, Open.

### Rewrite(FileVar, FileTitle)

Rewrite creates a file on disk with the name specified by FileTitle. This is the physical file name. FileTitle may be either a string constant or a variable of type string. If a file with the name specified by file title already exists it is destroyed and a new file with that name is created. The file created by the Rewrite statement is associated with file variable specified by FileVar, the logical file name. From this point on all references to the file will use the logical file name. After the Rewrite is executed, the file buffer is positioned to record number zero of the file. A file that is opened with the Rewrite statement is write only, meaning information can only be inserted into the file but not retrieved from it. Files that are opened using the Rewrite statement can only be accessed in a sequential manner.

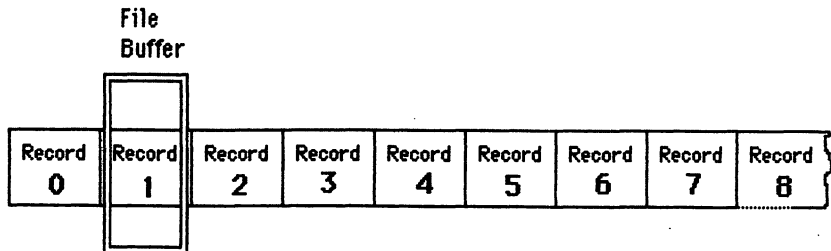


Figure 11-4.

### Reset(FileVar [ , FileTitle ])

Reset opens the disk file specified by the FileTitle, the brackets mean that it is optional. The disk file is assumed to already exist,

otherwise an error condition will exist and an error message will be displayed in an alert box. Like Rewrite, the FileTitle can be either a string constant or a string variable and the file variable FileVar is associated with the open file. Files that are open with Reset are only accessible for reading and can only be accessed sequentially. After a reset, the file buffer for the file is positioned to record number zero of the file and its contents are placed in the file buffer. A Reset with the optional parameter FileTitle omitted may also be used to get the zeroth record of a previously opened file into the file buffer and place the file buffer over record number zero in the file.

## Accessing Files

Data is written to and read to from a file with the Get and Put procedures. Get takes information from the file on disk and places it into the file buffer.

For files to be useful it is necessary to put values into files. To do that Pascal provides the Put statement. Put has the following syntax

```
Put(FileVar);
```

The Put statement puts the value contained in the file buffer variable, FileVar, into the location in the file that the file buffer points to. The file buffer is then advanced to the next location in the file. The file buffer variable must contain some defined value before it is placed into the file. The put statement cannot be used with a file that has been opened with a Reset.

The following section of code uses the Put statement to create a file containing the integers from ten down to one.

```
Rewrite(Numbers, 'Space Ship');
for Countdown := 10 downto 1 do
begin
  Numbers^ := Countdown;
  Put(Numbers)
end;
```

The logical file name is the file Numbers we declared before with components of integers. The Rewrite procedure creates a file on the disk called Space Ship and equates the logical and physical names for the file. The for loop assigns the value of the control



variable Countdown to the file buffer and then places that value into the file with the Put procedure.

After this loop executes the file will contain:

Record 0 contains 1

Record 1 contains 2

Record 2 contains 3

.

.

Record 9 contains 10

Once there is information in a file it can be accessed with a Get statement.

**Get( FileVar )**

The Get statement places the file buffer associated with FileVar over the next record in the file. The Get statement then places the contents of the record into the file buffer. The following program segment and diagrams show how file access with Get works.

```
var
  Numbers : file of integer;
  :
  :
  Reset(Numbers, 'Space Ship');
  Get(Numbers);
  Writeln(Numbers^);
  Get(Numbers);
  Writeln(Numbers^);
  :
  :
```

Once a record has been read into the file buffer variable that variable can be referenced just like any other variable. In the above case the file buffer variable Numbers^ is being used just like an ordinary integer would be used in the Write statement.

The first Get statement advances the file buffer so that it is over record 1 in the file. Get then places the record under the file buffer (record 1) into the file buffer. The second Get statement advances the file buffer over record two in the file and places the contents of that record into the file buffer (Figure 11-5).

## Closing a File

```
Close(FileVar);
```

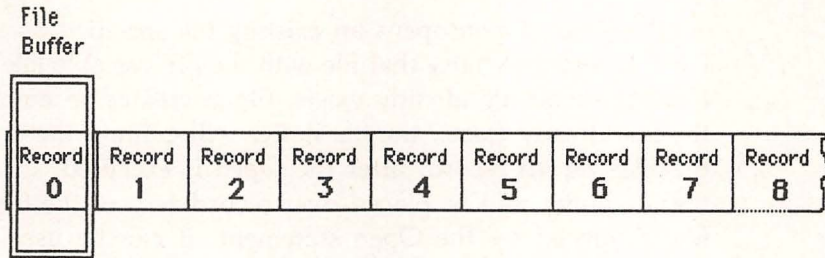


Figure 11-5.

The last part of using a file is closing it. Closing a file terminates the association between the logical file and the physical file. All subsequent access to the file buffer variable will produce an error because the file no longer exists.

## Mixing Get and Put

Most programs that do file I/O require both getting and putting of data in the same program. In order to do this the file must be opened with `rewrite`, accessed with `put`, closed, opened with `reset`, accessed with `get` and then closed, and so on. This is a very cumbersome process. In addition, since the files are sequential it takes almost an entire pass over the file to look at a record near the end of the file, and almost another entire pass to change the contents of a record near the end of the file. This is the scourge of using sequential files. It would be very convenient if it were possible to get or put any record we pleased without having to access all others.

## Using Random Access Files

Random access files provide a means of both using `Get` and `Put` statements alternately without having to open and close the file. The random access technique also allows a record anywhere in the file to be accessed directly without accessing any other records first. For this reason random access files are also sometimes known as direct access files.

To open a random access file the `Open` statement is used.

`Open(FileVar , FileName)`

Like `Reset`, `Open` opens an existing file specified by the string `FileTest` and associates that file with the file variable `FileVar`. Like `Rewrite`, if no file already exists, `Open` creates an empty file. If the file already exists, the file buffer will contain the contents of the file's zeroth record, after the `Open` is executed. In any case, the file buffer will be placed over record zero of the file. Once a file is opened by the `Open` statement, it can be used for both reading and writing. Unlike `Reset` and `Rewrite`, the `Open` statement allows files to be randomly accessed. One advantage of `Open` is that `Gets` and `Puts` can be mixed.

## Seek

### `Seek(FileVar,RecordNumber)`

Another advantage of opening a file with `Open` is that the file buffer can be pointed to any component in the file. The `Seek` statement points the file buffer associated with the file specified by `FileVar` to the record indicated by the integer value or constant specified by `RecordNumber`. The contents of that record are placed into the file buffer variable. `Seek` is used to point the file buffer to a specific record in a file before an access by a `Put` or `Get`. In actuality, the use of `Seek` precludes the need to do a `Get` afterwards because `Seek` will perform the role of the `Get` in placing the record into the file buffer variable.

As an example of a typical use of a `Seek` statement, let's go back to our file of integers and change the value of all the records we have placed in the disk file named `Space Ship`.

```

:
  Open(Numbers,'Space Ship');
  for RecordOfNumber := 0 to 9 do
begin
  Seek(Numbers, RecordOfNumber)
  {file buffer now contains record to be changed }
  {Change the value}
  Numbers^ := Number^ + 1;
  { put the changed record in the file buffer back into the
    file }
  Put(Numbers);
end;
:
:
```



## Finding the End Of a File

The above program goes through a file whose size is assumed to be known in advance (NumberOfRecords). It is not always the case that the size of a file can be known in advance. A business is not likely to know the number of customers or transactions that are necessary to be stored in advance. Therefore, the size of a file usually changes dynamically as needed. If a program that changes all the records in a file is run on such a file, then it is necessary to determine, while the program is running, when the end of the file has been reached. The built-in function Eof(FileVar) returns True if a Get or Seek has not attempted to access past the end of a file and False otherwise. We can use this to determine if the record we are attempting to read is contained in the file. The above program could be written as follows for a file of unspecified size.

```

Open(Numbers, 'Space Ship');
while not Eof(Numbers) do
  begin
    Seek(Numbers, RecordOfNumber)
    {file buffer now contains record to be changed }
    {Change the value}
    Numbers^ := Number^ + 1;
    { put the changed record in the file buffer back into
      the file }
    Put(Numbers);
  end;
  :
  :

```

Since the Eof function returns FALSE if a record exists and a while loop iterates while the condition is TRUE we must use a not to reverse the Boolean value returned by the function. The EOF function can also be used to read data from the keyboard. The Enter key on the keyboard is used to signal the end of file condition.

## Text Files

In addition to using files to store information on secondary storage devices, files are also useful for transferring information

to input/output devices. All along we have been using files for this purpose without knowing it. Every time we have used a Write statement, we have used a predeclared file called "Output" that is automatically opened for writing to the display screen. Each time we used a Read or Readln we read from a predeclared file called "Input" that is automatically opened for reading from the keyboard. A Write statement may have an additional parameter indicating what file the Write statement is writing to. The following two statements are equivalent.

```
Write('ABC');  
Write(Output, 'ABC');
```

The following two read statements, with and without a file parameter, are equivalent.

```
Read(Ch);  
Read(Input, Ch);
```

The predeclared files Input and Output are files of a predeclared type Text. The type Text is essentially the same as a file of characters with the exception that certain standard procedures that cannot be used with other types of files can be used with Text files. These standard procedures include Read, Readln, Write, Writeln and EOLN.

In addition to reading or writing to the default file devices it is also desirable to be able to Write to devices such as the printer. To send information to the printer rather than to the screen, a predeclared device file called 'Printer:' must be used. This file must be opened for writing using a Rewrite statement. The file variable associated with it must be declared to be of type Text. Reads and gets should not be used with the printer as the printer is an output only device.

```
var  
  Prnt : Text;  
  :  
begin  
  Rewrite(Prnt, 'Printer:');  
  Writeln(Prnt, 'This will print on the printer');
```

In your programs you may find it convenient to be able to direct the output from your program to either the Text window or your printer. This can be done by using a procedure to direct the output to the desired output device.

```

var
    Ch : Char;
    Output : Text;
    :
procedure SelectDevice;
begin
    Writeln('Where should the output go');
    Writeln('Enter P for the printer or T for the Text
        window');
    Read(Ch);
    if Ch = 'P' then
        Rewrite(Output, 'Printer:');
    if Ch = 'T' then
        {Do nothing}
end;
    :

```

If the user selects output to the printer, the Text file is associated with the printer by use of the Writeln statement. If output to the Text window is desired nothing need be done.

The Writeln statements in the program should all look like:

```
Writeln(Output, variable list);
```

There is another device file that can be associated with a file variable of type text on the Macintosh. 'Modem:' works like 'Printer:' except it refers to the modem port on the back of the Macintosh. 'Modem:' can be read from and written to and therefore should be opened using an Open. Naturally using the Seek makes no sense when used with a device.

Textfiles can also be used with disk files. In fact all programs written in Pascal are stored as textfiles on disk. The following program prompts the user for the name of a text file, (perhaps a MacPascal program file) and reads the file from the disk line by line, displaying each line in the text window.

```

program DisplayText;
var
    TextFile : Text;
    FileLine : string[80];
begin
    Reset(TextFile, OldFileName('Select a Textfile'));
    while not Eof(TextFile) do
        begin

```



```

    Readln(TextFile, FileLine);
    Writeln(FileLine)
end
end.

```

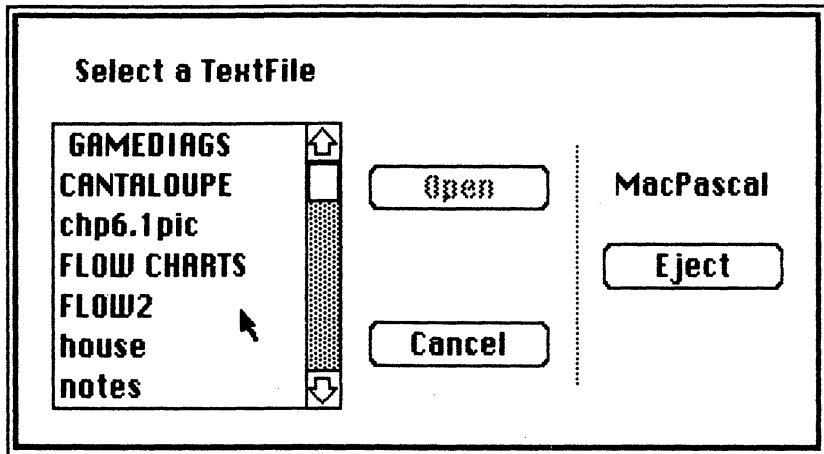


Figure11-6.

In this program, instead of using a string for the file's name in the reset statement, we used the toolbox function `OldFileName`. This function displays a dialog box on the screen with all the names of the files already on the disk. You can customize the dialog box by passing a string parameter with the desired prompt. This is the same routine that MacPascal uses when you open a MacPascal program to work on. MacPascal also provides a function `NewFileName` that works the same way except that it allows for the entry of a new file name. This is the same procedure MacPascal uses when you select **Save As..** from the **File** menu.

To illustrate the use of files let's look at a program to retrieve the phone number of a person given the first and last name. The program should have several functions:

1. Searching for a name and displaying information.
2. Adding a name.
3. Deleting a name.

One approach to writing this program might be to keep the entire file of names and numbers in memory in an array and do the searching, adding, and deleting of names to the array already in memory.

```
    Read all friends' names into array in computer's memory
repeat
    show menu of choices
    if choice is find then
        find a friend's name and number in the array
    else if choice is add then
        add a friend's name and number in the array
    else if choice is delete then
        delete a friend's name and number in the array
    else
        it's time to quit
until it's time to quit
Write all friends' names from memory to disk
```

The above algorithm has an advantage over other possible implementations in that all data is in memory and can be accessed very rapidly. There is a major disadvantage to this method also. When keeping all data in memory the amount of data that can be handled is limited to the size of the computer's memory. Another disadvantage is that since all the data is kept in the computer's memory during the program's execution, if power to the computer were lost all information in the computer's memory would also be lost.

Another method of implementing this would be to add, delete, and search directly to the data in the disk file. In this case the English-like pseudocode would remain the same with the exception of eliminating the reading into memory at the beginning and the writing to disk at the end. Let's look at the pseudocode for this version.

```
repeat
    show menu of choices
    if choice is find then
        find a friend's name and number in the file
    else if choice is add then
        add a friend's name and number in the file
    else if choice is delete then
        delete a friend's name and number in the file
    else
        it's time to quit
until it's time to quit
```

The implementation of this algorithm is straight forward and

simple to follow. There are however several features worth noting.

When the program is run for the first time, the file `Friends` does not exist on disk and must be created and initialized. After the program has been run once, the file already exists and only needs to be opened. In this program we take advantage of the way the `Open` procedure works. If the `Open` procedure is used and a file already exists, then that file is all set for use and nothing more need be done. If the `Open` procedure is used and no file already exists, an empty file is created. Having an empty file (containing no records) is not sufficient for this program. This program requires the file to be initialized with records. We can tell if the file is empty by using the `eof` function. If the file is empty `eof` returns true and we can initialize the file. The following program segment illustrates the creation of a file only when needed.

```
Open(Friend, 'Friend');
if Eof(Friend) then
    { The file does not already exist. Initialize the file }
else
    { The file already exists and is open and ready for use }
```

All searches in this program are done sequentially from the beginning of the file to the end of the file. A template for this kind of search through a file is illustrated by the following program segment.

```
Count := 0; { point to zero'th record of file }
Seek(Friend, Count);
while not Eof(Friend) do
begin
    { Check here to see if current record is one you want }
    { and perform appropriate action }
    Count := Count + 1;
    Seek(Friend, Count)
end
```

It is necessary to be able to tell if a record in the file is vacant or in use. We will do this by assigning the value `Empty` to the last name if the record is not being used. In this program the constant `Empty` corresponds to the null string (`""`). We delete records by marking them empty with the null string.

Here is the entire program. Type it in and try it out. The con-



stant `FileSize` can be changed if you want to hold more than ten names.

```

program PhoneBook;
{ Program to retrieve a phone number for any name }
{ contained in the data file }
const
    FileSize = 10;
    Empty = "";
type
    NameType = string[15];
    PhoneType = string[10];
    FriendType = record
        LastName : NameType;
        FirstName : NameType;
        PhoneNum : PhoneType
    end;
var
    Choice : Char;
    Friend : file of FriendType;
    First, Last : NameType;
    Phone : PhoneType;
    Location : Integer;
    TextWindow : Rect; { Hold dimensions of text window }
procedure Menu;
begin
    Page(Output);
    Writeln('1. Search for Name');
    Writeln('2. Add a Name');
    Writeln('3. Delete a Name');
    Writeln('0. Quit');
    Writeln;
    Write('Please Choose > ');
    Read(Choice);
    Writeln;
    while not (Choice in ['0'..'3']) do
        begin { Not a valid Choice }
            SysBeep(5);
            Read(Choice)
        end
    end;
procedure InitFile;

```

```
{ Create an empty file of Friends }
var
    Count : Integer;
begin
    for Count := 0 to FileSize do
        begin
            Seek(Friend, Count);
            with Friend^ do
                begin
                    FirstName := Empty;
                    LastName := Empty;
                    PhoneNum := Empty;
                end;
            Put(Friend)
        end;
    Close(Friend); { Making the file permanent on disk }
    Open(Friend, 'Friends') { Open it again, for further use }
end;
procedure GetName;
begin
    Write('Enter Last Name > ');
    ReadLn>Last);
    Write('Enter First Name > ');
    ReadLn(First)
end;
procedure GetPhone;
begin
    Write('Enter Phone Number > ');
    ReadLn(Phone)
end;
function FindFriend (First, Last : NameType;
                    var Count : Integer) : Boolean;
{ Returns true if the name in First, Last is found. }
{ Returns false if the name is not found. If the }
{ name was found, Count contains its file position }
var
    Found : Boolean;
begin
    Count := 0;
    Found := False;
    Seek(Friend, Count);
    while (not Eof(Friend)) and (not Found) do
```

```
begin
  with Friend^ do
    if (FirstName = First) and (LastName = Last) then
      begin { Found a match }
        WriteLn(FirstName, ' ', LastName, ' ',
          PhoneNum);
        Found := True
      end
    else
      begin { No match look at next record }
        Count := Count + 1;
        Seek(Friend, Count)
      end
    end;
  FindFriend := Found
end;

procedure AddFriend (First, Last : NameType;
  Phone : PhoneType);
{ Adds the Friend whose name is passed in to the file }
var
  Added : Boolean;
  Count : Integer;
begin
  { Look for a record with no last name }
  Added := False;
  Count := 0;
  Seek(Friend, Count);
  while (not eof(Friend)) and (not Added) do
    if Friend^.LastName = Empty then
      begin { found an empty spot }
        Seek(Friend, Count);
        Friend^.LastName := Last;
        Friend^.FirstName := First;
        Friend^.PhoneNum := Phone;
        Put(Friend);
        Added := true
      end
    else
      begin { look at next spot }
        Count := Count + 1;
        Seek(Friend, Count)
      end;
end;
```



**if not Added then**

    Writeln('File is full, Press <Return> to continue')

**end;**

**procedure** Delete (Location : integer);

{ Deletes the record specified by Location by putting}  
{ an Empty string in that location }

**var**

        Which : Char;

**begin**

        Seek(Friend, Location);

        Write(' Delete (Y/N) > ');

        Read(Which);

**if** (Which = 'Y') **or** (Which = 'y') **then**

**begin** { Record is deleted by setting all fields to Empty }

                Seek(Friend, Location);

                Friend^.LastName := Empty;

                Friend^.FirstName := Empty;

                Friend^.PhoneNum := Empty;

                Put(Friend);

**end**

**end;**

**begin**

        { Setup the Screen }

        HideAll;

        SetRect(TextWindow, 40, 40, 300, 300);

        SetTextRect(TextWindow);

        ShowText;

        Open(Friend, 'Friends');

**if** Eof(Friend) **then**

            InitFile; { file doesn't already exist, create it }

**repeat**

        menu; { Show the choices }

**case** Choice **of**

            '0' :

            ; { Do Nothing }

            '1' : { Search for a Friend }

**begin**

                Page(output);

                GetName;

**if not** FindFriend(First, Last, Location) **then**

**with** Friend^ **do**

```
        Writeln(FirstName, ' ', LastName, ' was not
                found. ');
        Writeln('Press <Return> to continue. ');
        Readln
    end;
    '2' : { Add a Friend }
    begin
        Page(output);
        GetName;
        GetPhone;
        AddFriend(First, Last, Phone)
    end;
    '3' : { Delete a Friend }
    begin
        Page(output);
        GetName;
        if FindFriend(First, Last, Location) then
            Delete(Location)
        end
    end { case }
until Choice = '0';
Close(Friend)
end.
```

# 12 Variant Records and Pointers

In this chapter we will discuss two of the more sophisticated and useful features in Pascal. Variant records are an extension of the concept of records seen in Chapter 10. Pointers are a data type totally different than any other we have seen so far in Pascal.

## Variant Records

In some situations, several different records that only differ slightly are needed. For example, suppose we want to represent information about all the people who are on campus at a college. For each we would want to keep a name, address, and identification number. But depending upon whether an individual was a student or a faculty member we would want to maintain different information. For students, we want to include their class standing, but for members of the faculty, we want to include their job title and what department they work for. With variant records this different information can be represented in a single record structure. Variant records allow the value of one field in the record to determine the type of information that can be stored in other fields.

```
type
  Person = (Student, Faculty);
  InfoRec = record
    Name : string[20];
    IDNum : string[9];
```



```

end; {InfoRec}
case Occupation : Person of
  Student :
    (Standing : 1..4);
  Faculty :
    (Description : string[20];
     Department : string[20])
end;

```

VARIANT RECORD

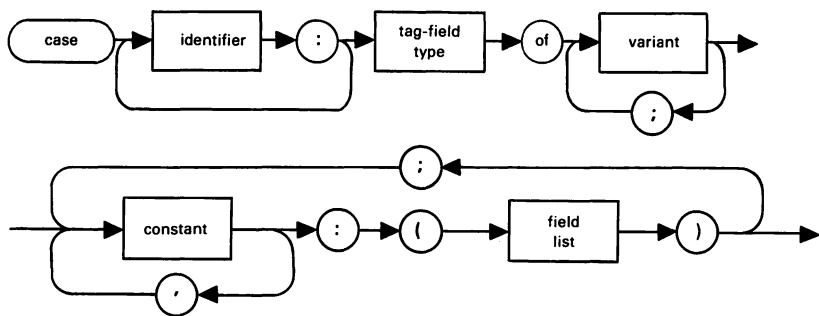


Figure 12-1.

The declaration of a variant record differs from the standard record declaration because a **case** statement is included. The **case** statement must be the last statement in the declaration and the fields listed above it are handled just like a standard record declaration. The **case** statement is used to decide which fields are to be included in the record depending upon the value of the tag field (in this situation, Occupation is the tag field). The tag field is included in all variations of the record but depending on the value of that field, one of the groups of fields listed after the possible values are included. This determination is made during the execution of the program and which of the variant fields included will change if the value of the tag field changes. Notice that there is no **end** used for the **case** statement. Let's declare two records to be of type InfoRec and examine them more closely.

```

var
  Person1, Person2 : InfoRec;

```

```

Person1.Name := 'Marian';
Person1.IDNum := '12355321';
Person1.Occupation := Student;
Person1.Standing := 4;

```

In the record Person1 the tag field Occupation is set to Student. The variant field that is included after that is Standing.

```

Person2.Name := 'Alan';
Person2.IDNum := '12351234';
Person2.Occupation := Faculty;
Person2.Description := 'Assistant Professor';
Person2.Department := 'Computer Science';

```

In the record Person2 the tag field is set to Faculty. The variant fields that are included are Description and Department. Notice that the number of fields in the variant part do not have to be the same. Variant parts could also be declared containing no fields at all. The fields selected by the Case statement are called the active fields. For example suppose the value of Person2.Occupation (the tag field) is changed to Student. It would then be illegal to reference the Person2.Description because it is not active.

It is not uncommon for a Pascal programmer to use a variant record without a tag field. This is legal to do. Instead of the tag field just a tag type is used in the Case statement.

```

TestRec = record
  case Boolean of
    True : (Int : Integer);
    False : (Ch : Char)
  end; {record}

```

When a variant record is declared in this fashion all the variant fields are active at the same time. However, only one field is actually created. This technique can only be used when the programmer knows what will be stored in the field from the context that the record is used in. An example of this would be when a programmer knows that all even records in a file will contain a number and all odd records will contain a character, or any other similar scheme. The advantage is that a significant amount of space is saved because both fields do not have to be included in each record.

Variant records are a space saver because one file can be used to hold several different types of records. The space taken up by

a variant record is the space needed to hold the largest combinations in the variant records. This is done because all records must be the same size so that file operations can be done. In memory the different variable parts occupy the same memory locations. Therefore, if one variant field is active and contains a value and then a different variant field becomes active, the original value is still available to be used. This allows tricks to be done that circumvent Pascal's type checking. It is recommended that this technique not be used unless you are familiar with the way MacPascal represents data internally.

## Pointers

All the variables we have seen so far have been static variables. This meant that the variables were declared in the Var section of a program, and memory space was allocated when the program started to execute. When we use static variables we must know in advance how much data we will need to store. For example, if we use an array to keep track of all the students in a university we would have to know in advance the maximum number of students that may register and then declare that many elements in the array. If half the students decided to drop out after taking Computer Science 101 last semester, then half the array would be wasted. On the other hand, if more people enrolled than anticipated, the array would be smaller than required and the program would not work.

In Pascal, we have a way of creating new variables dynamically as they are required during program execution. These dynamic variables are not accessed the way static variables are, but rather through other variables known as pointers. To show how pointers work let's look at a simple example of a dynamic Integer and a pointer to it.

```
var  
  P : ^Integer;
```

This declares P as a pointer to an integer variable. Notice that a pointer is declared with an up arrow ^ (on top of the 6 key on the Macintosh) preceding the data type of the variable it will point to. This declaration declares only the pointer not the variable. We create the variable that P will point to dynamically with the New procedure.



New(P);

This procedure creates an Integer variable that is not named but can be referenced through the pointer P. The variable P points to can be accessed with P $\wedge$ .

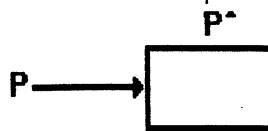


Figure 12-2.

P            The pointer

P $\wedge$         How we reference the variable P points to

When a dynamic variable is created its value is uninitialized. A value can be placed in the newly created variable with :

P $\wedge$  := 26;

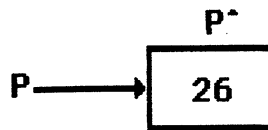


Figure 12-3.

P $\wedge$  refers to the variable and can be used like any other integer variable.

The statement

P := 26;

has no meaning and will cause an error since P only points to an integer and can't contain a value itself. If a second pointer to an Integer Q were declared we could make it point to the same variable that P does with:

Q := P;

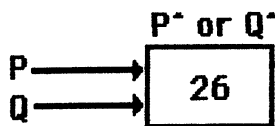


Figure 12-4.

Q now points to the same variable that P does.

An assignment statement with pointers takes on a slightly different meaning than with any other type of variable. The pointer on the left side of the assignment operator is made to point to the same variable as the one on the right side of the assignment operator. We can access the variable we created with either of the two pointers that point to it, either Q or P.

```
Q^ := 4;  
Write(P^);
```

This prints the value of P^ which is 4.

When a variable created dynamically is no longer needed we can dispose of it and liberate the memory locations it occupied with the Dispose procedure.

```
Dispose(P);
```

This destroys the variable that P pointed to. When a pointer doesn't point to a variable it is said to point to NIL. NIL is a Pascal reserved word and a predeclared constant. The link between a pointer and its variable can be destroyed by assigning NIL to it.

```
P := NIL;
```

So far, what we have seen as the use of pointers has no advantage over the use of static variables since we must declare a pointer for every dynamic variable used. Hence we are in the same boat as before, having to know the number variables that will be created prior to program execution. The advantage to using pointers can be seen when a dynamic variable contains a pointer to another dynamic variable. This can be done with records.

```
type  
  Dynamic : record  
    Data : Integer;  
    Link : ^Dynamic  
  end;
```

The declared record type has two fields: Data which will contain an Integer; and Link which is a pointer to the type Dynamic (the record type itself). Now let's declare two records of type Dynamic.

```
var  
  P, Q : ^Dynamic;
```

P and Q are both pointers to the record type dynamic. Notice that no records actually exist at this time, only pointers to records. We can create a record dynamically with :

`New(P);`

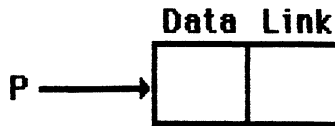


Figure 12-5.

A record now exists and P points to it. A second record can be created with:

`New(Q);`

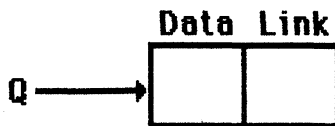


Figure 12-6.

We now have two dynamic records pointed to by P and Q. These two records can be linked together by connecting the pointer field ( $P\wedge\text{Link}$ ) of the first record to the second record. This is done by making it point to the same thing that Q does.

`P $\wedge$ .Link := Q;`

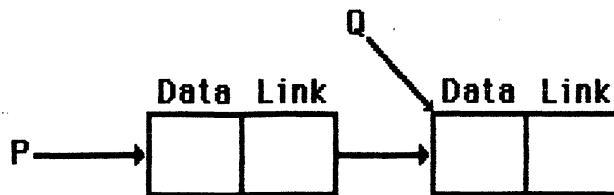


Figure 12-7.

A third record can be added to our chain of records by creating a new record that Q points to and then linking it to the second record.

`New(Q);`  
`P $\wedge$ .Link $\wedge$ .Link := Q;`



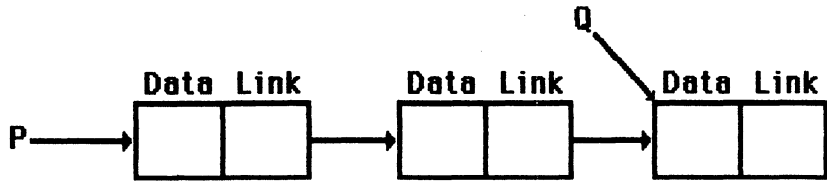


Figure 12-8.

We were able to use  $Q$  to create a new record even though it already pointed to something. The link to what it previously pointed to is lost and  $Q$  will point to a newly created record. The complicated  $P\wedge\text{Link}\wedge\text{Link}$  refers to the Link field of the second record. This is like an expression evaluating to a field. It is evaluated left to right.

|                                       |   |
|---------------------------------------|---|
| $P\wedge\text{Link}$                  | Refers to the Link field of the first record, the one that $P$ points to.       |
| $P\wedge\text{Link}\wedge$            | Refers to what the link field of the first record points to, the second record. |
| $P\wedge\text{Link}\wedge\text{Link}$ | Refers to the Link field of the second record.                                  |

A chain of records linked together like this is called a linked list. A linked list is a dynamic data structure which has similar uses to the static data structure, arrays. The disadvantage of using a link list is that records in the list can not be accessed without tracing through the list. It is essentially a sequential access structure. A quicker and more efficient way to create a linked list uses a for loop and three pointers.

```

var
  P, Q, R : ^Dynamic;

```

```

.
.
New(P);
R := P;
for I := 1 to 3 do
  begin
    New(Q);
    R^Link := Q;
    R := Q
  end

```

In the earlier example two pointers were used, one to point to the first record in the list and the second to create new dynamic

records. When using the `for` loop we need three pointers, one to point to the first record (P), one to create new dynamic records(Q), and a third to point to the last record in the list(R). The use of the third pointer alleviates the need to spell out the name of the Link field in the last record like we had to do in the other example. Instead since R points to the last record, `R^.Link` refers to the pointer we must link to the newest record.

Let us now look at a real application for pointers. Suppose we wanted to create a program that is similar to the program `DisplayText` in Chapter 11. Unlike `DisplayText`, which printed a file from beginning to end, the new program `ReverseText`, will display the file backwards from end to beginning. Since the file we wish to print is a text file, it can only be accessed sequentially. Therefore, it can only be read from beginning to end. We will have to read the entire file into memory before we can start printing it out in reverse order. We do not know the size of the file to be reversed in advance, so we will use a dynamic variable to hold each line of the file as it is read in.

The variable that holds each line of text to be read in will be the record `LineRec` which contains two fields. The field, `TextLine`, is a string that holds a line of text read in, and the other field, `Previous`, is a pointer that points to another record of the same type (the record it points to holds the previous line of text read in).

**type**

`LinePtr = ^LineRec;`

`LineRec = record`

`TextLine : string[80]; { hold a line of text }`

`Previous : LinePtr { pointer to the previous line }`

`end;`

Notice that the type `LinePtr` is declared to be a pointer to the type `LineRec` which has not yet been declared. This is the only situation in Pascal that allows an identifier to be referenced before it has been declared.

The program works by reading the first line in to a newly created dynamic variable. Since there is no previously read in line for the previous field to point to, we make it point to `Nil`. We then continue to read lines from the file into a dynamically created variable making each `Previous` pointer point to the previous line read in. When there are no more lines to be read in, we can trace through the linked list created in the reverse order, since

each pointer points back to the previous line, and prints out each line.

```
program ReverseText;
{ program to reverse a textfile of arbitrary size using
  pointers }
type
  LinePtr = ^LineRec;
  LineRec = record
    TextLine : string[80]; { hold a line of text }
    Previous : LinePtr { pointer to the previous line }
  end;
var
  TextFile : Text;
  LineBefore, NewLine : LinePtr;
begin
  Reset(TextFile, OldFileName('Select a Textfile'));
  LineBefore := NIL;
  while not Eof(TextFile)do
    begin
      New(NewLine);
      Readln(TextFile, NewLine^.TextLine);
      NewLine^.Previous := LineBefore;
      LineBefore := NewLine
    end;
    { Trace through linked list in reverse order printing
      lines }
  while LineBefore <> NIL do
    begin
      Writeln(LineBefore^.TextLine);
      LineBefore := LineBefore^.Previous
    end
  end.
```



# 13 A Look At An Application—The Checking and Savings Program

This chapter has been placed at the end of the book for a reason. After reading the entire book you should be familiar with all the aspects of Pascal programming. However, after learning how to program in Pascal there is still one more skill to be learned. That is how to plan and program a significant project. In this chapter, we will write a program that will prove useful in your banking transactions.

## Overview

The Checking and Savings program will track transactions for both a checking account and a savings account. These were both included because many banks now bundle a checking and savings account together. Several types of transactions can be entered for both accounts. For the checking account the transactions that can be entered are:

1. deposits
2. checks written
3. interest paid by the bank (for interest paying checking accounts)
4. fees charged by the bank

For the savings account the transactions accepted are:

1. deposits
2. withdrawals
3. interest payed by the bank

For either account a report of all the transactions can be printed with the current balance.

## Data Structures

One of the first things to consider when developing any application is what data structure will be used. That is, how will the data be organized, handled and stored. In this program we essentially have two types of information. The first is a transaction for either checking or savings. Since the transactions for both types of accounts are similar they can both be stored in the same record structure with a field indicating which account the transaction is for. The other fields in the record are needed to keep track of the particulars of the transactions. The record structure is:

```

type
  TransType = (Checking, Savings);
  TransRec = record
    CheckOrSave : TransType;
    Code : 1..5;
    Date : string[8];
    Amount : Integer;
    TaxDeduct : Char;
    CheckNumber : Integer;
    PayTo : string[80]
  end;
```

The first four fields are needed for any type of transaction. The last three are used only for a check. After a transaction is entered it is written to an external data file named Trans.data. The field Code is used to specify which type of transaction the record represents. The codes are:

| Code | Checking   | Savings  |
|------|------------|----------|
| 1    | Deposit    | Deposit  |
| 2    | Withdrawal | Check    |
| 3    | Interest   | Interest |
| 4    | Fee        | --       |

The second Type of information to keep track of is the balance of both accounts. We could not include that as a field in the transaction records since the balance changes after each transaction. A separate record structure is used to hold the balances.

```
BalanceRec = record
  SavBal : Integer;
  CheckBal : Integer
end;
```

A different field in the record is used for the balance in each account. This record is updated after every transaction. At the end of the program this record is written to an external disk file named Balance.data. When the program is run subsequent times, the record is read from this file. In this way up-to-date balance information is maintained.

## Development

This program differs from all others we have seen in the book because several program options are available to the user and all the options are not done in any specific order. The options are selected by the user from a series of menus. The selection entered to these menus controls the execution. The main menu provides the choices of a savings or checking transaction to be entered, balances to be displayed or exiting from the program. The checking or savings choices lead to sub-menus which provide the choice transactions. The balance options display the balances and the exit option does the file housekeeping before returning control of MacPascal to the user. A first level of development for the program would produce pseudocode indicating what action will take place for each of the possible program options.

```
repeat
  Display Main Menu
  Get option
  Case option of
    1 : Savings transaction
    2 : Checking transaction
    3 : Display balances
    4 : Exit program
  until Exit option is picked
```



A second level of development will show detailed pseudocode for how each of the main menu options will be processed.

### SAVINGS TRANSACTIONS

Deposit

Get information

Add amount to savings balance

Place transaction in file

Withdrawal

Get information

Subtract amount from savings balance

Place transaction in file

Interest credited

Get information

Add amount to savings balance

Place transaction in file

Report

while not (eof(Trans.data))

begin

Read a transaction

If savings transaction then Print transaction  
information

end {while}

Print balance

### CHECKING TRANSACTIONS

Deposit

Get information

Add amount to checking balance

Place transaction in file

Check

Get information

Subtract amount from checking balance

Place transaction in file

Interest credited

Get information

Add amount to checking balance

Place transaction in file

Checking Fee

Get information

Subtract amount from checking balance

Place transaction in file

Report

```

    while not (eof(Trans.data))
        Read a transaction
        If checking transaction then Print transaction
            information
        Print balance
    DISPLAY BALANCES
    Clear screen
    Display balances
    EXIT
    Replace balance record in Balance.data
    Close all files

```

A third level of refinement will include writing the main program with the variable declarations and the procedures that implement the menu structure.

```

program CheckingAndSavings;
type
    TransType = (Checking, Savings);
    TransRec = record
        CheckOrSave : TransType;
        Code : 1..5;
        CheckNumber : Integer;
        PayTo : string[60];
        Date : string[8];
        Amount : Real;
        TaxDeduct : Char;
    end;
    BalanceRec = record
        SavBal : Real;
        CheckBal : Real;
    end;
var
    Out : Text;
    Transaction : TransRec;
    Balance : BalanceRec;
    FourSet, FiveSet : set of 1..10;
    Option : Integer;
    TransFile : file of TransRec;
    BalanceFile : file of BalanceRec;
    Stop : Boolean;
    I : Integer;
begin {Main program}

```

```

Stop := False;
FourSet := [1,2,3,4]; {The sets are used for input
                        verification}
FiveSet := [1,2,3,4,5];
Open(BalanceFile, 'Balance.Data');
Rewrite(Out, 'Printer:');
if eof(BalanceFile) = True then
  begin
    Balance.SavBal := 0;
    Balance.CheckBal := 0
  end {Read Balance File}
else
  Balance := BalanceFile^;
  {OpenTransaction File}
  Open(TransFile, 'Trans.data');
  Seek(TransFile, MaxInt); {move to last position in file}
  repeat {main driver}
    DisplayMainMenu;
    InitRec;
    case option of
      1:
        begin
          Transaction.CheckOrSave := Checking;
          CheckingMenu;
          CheckingOptions
        end;
      2:
        begin
          SavingsMenu;
          Transaction.CheckOrSave := Savings;
        end;
      3:
        ShowBalances;
      4:
        Stop := True;
    end; {case}
  until Stop = True;
  Close(TransFile);
  Seek(BalanceFile, 0);
  Balancefile^ := Balance;
  Put(BalanceFile); {Replace Balance record in
                    file}

```



```

    Close(BalanceFile);
end.{program}

```

The main program first initializes the variables used in the program and opens the files needed. If the balance file does not exist then the fields in its record are set to zero. The other function of the main program is to drive the program by calling a procedure to display the main menu and then calling the procedures necessary to handle the transaction to be entered. The procedures called from the main program are:

|                 |   |
|-----------------|---|
| DisplayMainMenu | Displays the main menu in the text window.  |
| InitRec         | Initializes the transaction record so that no information is in it from the previous transaction. |
| CheckingMenu    | Displays the menu of checking options.  |
| CheckingOptions | Processes the checking transactions.  |
| SavingsMenu     | Displays the menu of savings option.  |
| SavingsOptions  | Processes the savings transactions.   |
| ShowBalances    | Displays the current account balances.  |

The next step of the development is to write the procedures called in the main program. This set of procedures will handle all the various account transactions.

```

procedure DisplayMainMenu;
begin
    Page;
    Writeln('Checking and Savings System');
    Writeln;
    Writeln(' 1. Checking Transaction');
    Writeln(' 2. Savings Transaction');
    Writeln(' 3. Show Balances');
    Writeln(' 4. Exit');
    Writeln;
    repeat
        Write('Selection ');
        Readln(Option);
    until Option in FourSet;
end; {DisplayMainMenu}

```

This procedure displays the main menu. The user's selection is returned to the main program in the global variable Option.

Notice the input verification done in the Repeat loop which utilizes sets.

```

procedure CheckingMenu;
begin
    Page;
    Writeln('Checking System');
    Writeln;
    Writeln('1. Enter a Deposit');
    Writeln('2. Enter a Check');
    Writeln('3. Enter Interest');
    Writeln('4. Checking Fees');
    Writeln('5. Checking Report');
    repeat
        Writeln;
        Write('Selection ');
        Read(Option)
    until Option in FiveSet;
end; {CheckingMenu}
procedure SavingsMenu;
begin
    Page;
    Writeln('Savings System');
    Writeln;
    Writeln('1. Enter a Deposit');
    Writeln('2. Enter a withdrawal');
    Writeln('3. Enter Interest');
    Writeln('4. Savings Report');
    repeat
        Write('Selection ');
        Read(Option)
    until Option in FourSet;
end; {SavingsMenu}

```

The procedures CheckingMenu and SavingsMenu are very similar. They both display the possible account transaction and send the selected option back to the main program in the global variable Option.

```

procedure SavingsOptions;
begin
    case Option of
        1 :

```

```

    begin
        EnterDeposit;
        Balance.SavBal := Balance.SavBal +
            Transaction.Amount;
        WriteTransaction
    end;
2 :
    begin
        Enterwithdrawal;
        Balance.SavBal := Balance.SavBal—
            Transaction.Amount;
        WriteTransaction
    end;
3 :
    begin
        EnterInterest;
        Balance.SavBal := Balance.SavBal +
            Transaction.Amount;
        WriteTransaction
    end;
4 :
    PrintSavingReport;
end {case}
end; {SavingsOptions}
procedure CheckingOptions;
begin
    case Option of
    1 :
        begin
            EnterDeposit;
            Balance.CheckBal := Balance.CheckBal +
                Transaction.Amount;
            WriteTransaction
        end;
    2 :
        begin
            EnterCheck;
            Balance.CheckBal := Balance.CheckBal—
                Transaction.Amount;
            WriteTransaction
        end;
    3 :

```



```

    begin
        EnterInterest;
        Balance.CheckBal := Balance.CheckBal +
            Transaction.Amount;
        WriteTransaction
    end;
4 :
    begin
        EnterFee;
        Balance.CheckBal := Balance.CheckBal—
            Transaction.Amount;
        WriteTransaction
    end;
5 :
    PrintCheckingReport;
end {case}
end;{CheckingOptions}

```

The procedures SavingsOptions and CheckingOptions handle the account transactions. The selection entered to CheckingMenu or SavingsMenu is used as the selector in a Case statement. Several more procedures are called which accept the transaction data. The balance is then calculated and the transaction written to the file.

|                     |  |
|---------------------|--|
| EnterDeposit        | Prompts the user for deposit information.    |
| Enterwithdrawal     | Prompts the user for withdrawal information. |
| EnterInterest       | Prompts the user for interest information.   |
| EnterFee            | Prompts the user for fee information.        |
| EnterCheck          | Prompts the user for check information.      |
| WriteTransaction    | Write the transaction record to the file.    |
| PrintCheckingReport | Prints the report for the checking account.  |
| PrintSavingsAccount | Prints the report for the savings account.   |

The next step in the development is to write this set of procedures.

```

procedure EnterCheck;

```

```

begin
  with Transaction do
    begin
      Page;
      Writeln('Enter a Check');
      Writeln;
      Write('Check Number:');
      Readln(CheckNumber);
      Write('Paid to:');
      Readln(PayTo);
      GetAmount;
      Write('Tax Deductible(Y/N):');
      Read(TaxDeduct);
      Writeln;
      Code := 2;
      Write('Enter Date MM/DD/YY: ');
      Readln(Date);
    end
  end; {EnterCheck}
  procedure EnterDeposit;
  begin
    with Transaction do
      begin
        Page;
        Writeln('Enter a Deposit to ', Transaction.
          CheckOrSave);
        Writeln;
        GetAmount;
        Write('Enter Date MM/DD/YY: ');
        Readln(Date);
        Code := 1;
      end {with}
    end; {EnterDeposit}
    procedure EnterInterest;
    begin
      Page;
      Writeln(' Enter Interest to ', Transaction.CheckOrSave);
      Writeln;
      with Transaction do
        begin
          GetAmount;
          Code := 3;

```

```

        Write('Enter Date MM/DD/YY: ');
        Readln(Date);
    end {with}
end; {EnterInterest}
procedure EnterFee;
begin
    with Transaction do
        begin
            Page;
            Writeln('Enter a Fee');
            Writeln;
            GetAmount;
            Write('Enter Date MM/DD/YY: ');
            Readln(Date);
            Code := 4;
        end
    end; {EnterFee}
procedure EnterInterest;
begin
    Page;
    Writeln(' Enter Interest to ', Transaction.CheckOrSave);
    Writeln;
    with Transaction do
        begin
            GetAmount;
            Code := 3;
            Write('Enter Date MM/DD/YY: ');
            Readln(Date);
        end {with}
    end; {EnterInterest}
procedure EnterFee;
begin
    with Transaction do
        begin
            Page;
            Writeln('Enter a Fee');
            Writeln;
            GetAmount;
            Write('Enter Date MM/DD/YY: ');
            Readln(Date);
            Code := 4;
        end
    end

```

```

end; {EnterFee}
procedure EnterWithdrawal;
begin
    with Transaction do
        begin
            Page;
            Writeln('Enter a Withdrawal to ', CheckOrSave);
            Writeln;
            GetAmount;
            Code := 2;
            Write('Enter Date MM/DD/YY: ');
            Readln(Date)
        end {With}
    end; {EnterWithdrawal}

```

All of these procedures are very similar prompting the user for the transaction information and assigning it into the Transaction record. Because of the similarity between the two accounts the procedures EnterDeposit and EnterInterest are used for both checking and savings.

```

procedure PrintCheckingReport;
begin
    PrinterMessage;
    Writeln(Out, 'Checking Report');
    Seek(TransFile, 0);
    while not (eof(TransFile)) do
        begin
            if TransFile^.CheckorSave = Checking then
                with TransFile^ do
                    begin
                        Write(Out, Date);
                        case code of
                            1 :
                                Write(Out, 'Deposit');
                            2 :
                                begin
                                    Writeln(Out, 'Check number ',
                                        CheckNumber, ' : 10,
                                        'Deductible ', TaxDeduct);
                                    Write(Out, 'Paid to:', PayTO)
                                end;
                            3 :

```



```

        Write(Out, 'Interest ');
    4 :
        Write(Out, 'Fee');
    end; {case}
    Writeln(Out, Amount : 7 : 2);
end; {with}
Get(TransFile)
end; {While}
Writeln(Out);
Writeln(Out, 'Balance ', Balance.CheckBal : 7 : 2)
end; {PrintCheckingReport}
procedure PrintSavingReport;
begin
    PrinterMessage;
    Writeln(Out, 'Savings Report');
    Seek(TransFile, 0);
    while not (eof(transfile)) do
        begin
            if TransFile^.CheckorSave = Savings then
                with TransFile^ do
                    begin
                        Write(Out, Date);
                        case Code of
                            1 :
                                Write(Out, 'Deposit');
                            2 :
                                Write(Out, 'Withdrawal');
                            3 :
                                Write(Out, 'Interest ')
                        end; {case}
                        Writeln(Out, Amount : 7 : 2);
                    end; {with}
                Get(TransFile)
            end; {While}
        Writeln(Out);
        Writeln(Out, 'Balance ', Balance.SavBal : 7 : 2)
    end; {PrintSavingsReport}

```

Both of the report procedures read the transaction file starting at the first record. A Seek is used to position the file pointer to the zeroth record. Since the Seek reads the contents of the record it points to, no Get is necessary. Subsequent records are read

with a Get until the end of the file is reached. The CheckOrSave field in the record is examined, if the record is the proper type of transaction then the information is sent to the printer. A last set of procedures are called by this set.

**GetAmount**        Simply prompts the user for the amount of the transaction. This is needed in enough places to justify making it a procedure all its own.

**PrinterMessage**   Tells the user to turn on the printer.

```

procedure GetAmount;
begin
    Write('Amount: $');
    Readln(Transaction.Amount)
end; {GetAmount}
procedure PrinterMessage;
begin
    Page;
    Writeln('Set up printer');
    Writeln('Then press the mouse button');
    repeat
        {Do nothing loop}
    until button
end; {PrinterMessage}

```

The only line of interest in these two procedures is the Repeat loop in PrinterMessage. This loop is used to freeze the menu on the screen until the user presses the mouse button. This causes the built-in Boolean function Button to return True and the loop to end.

Here is the entire program together.

```

program CheckingAndSavings;
type
    TransType = (Checking, Savings);
    TransRec = record
        CheckOrSave : TransType;
        Code : 1..5;
        CheckNumber : Integer;
        PayTo : string[60];
        Date : string[8];
        Amount : Real;
        TaxDeduct : Char;
    end;

```

```

        BalanceRec = record
            SavBal : Real;
            CheckBal : Real
        end;
var
    Out : Text;
    Transaction : TransRec;
    Balance : BalanceRec;
    FourSet, FiveSet : set of 1..10;
    Option : Integer;
    TransFile : file of TransRec;
    BalanceFile : file of BalanceRec;
    Stop : Boolean;
    I : Integer;
procedure DisplayMainMenu;
begin
    Page;
    Writeln('Checking and Savings System');
    Writeln;
    Writeln(' 1. Checking Transaction');
    Writeln(' 2. Savings Transaction');
    Writeln(' 3. Show Balances');
    Writeln(' 4. Exit');
    Writeln;
    repeat
        Write('Selection ');
        Readln(Option);
    until Option in FourSet;
end; {DisplayMainMenu}
procedure CheckingMenu;
begin
    Page;
    Writeln('Checking System');
    Writeln;
    Writeln('1. Enter a Deposit');
    Writeln('2. Enter a Check');
    Writeln('3. Enter Interest');
    Writeln('4. Checking Fees');
    Writeln('5. Checking Report');
    repeat
        Writeln;
        Write('Selection ');

```

```

    Read(Option)
    until Option in FiveSet;
end; {CheckingMenu}
procedure SavingsMenu;
begin
    Page;
    Writeln('Savings System');
    Writeln;
    Writeln('1. Enter a Deposit');
    Writeln('2. Enter a Withdrawal');
    Writeln('3. Enter Interest');
    Writeln('4. Savings Report');
    repeat
        Write('Selection ');
        Read(Option)
    until Option in FourSet;
end; {SavingsMenu}
procedure InitRec;
begin
    with Transaction do
        begin
            CheckNumber := 0;
            PayTo := '';
            Amount := 0;
            TaxDeduct := '';
            Date := '';
        end
    end; {InitRec}
procedure WriteTransaction;
begin
    TransFile^ := Transaction;
    Put(TransFile)
end; {WriteTransaction}
procedure GetAmount;
begin
    Write('Amount: $');
    Readln(Transaction.Amount)
end; {GetAmount}
procedure PrinterMessage;
begin
    Page;
    Writeln('Set up printer');

```



```

        Writeln('Then press the mouse button');
        repeat
        {Do nothing loop}
        until button
    end; {PrinterMessage}
    procedure ShowBalances;
    begin
        Page;
        with Balance do
        begin
            Writeln('Checking: ', CheckBal : 7 : 2);
            Writeln('Savings: ', SavBal : 7 : 2);
            Writeln;
        end;
        Writeln('Hit the mouse button to continue');
        repeat
        until button
    end;
    procedure EnterCheck;
    begin
        with Transaction do
        begin
            Page;
            Writeln('Enter a Check');
            Writeln;
            Write('Check Number:');
            Readln(CheckNumber);
            Write('Paid to:');
            Readln(PayTo);
            GetAmount;
            Write('Tax Deductible(Y/N):');
            Read(TaxDeduct);
            Writeln;
            Code := 2;
            Write('Enter Date MM/DD/YY: ');
            Readln(Date);
        end
    end; {EnterCheck}
    procedure EnterDeposit;
    begin
        with Transaction do
        begin

```

```

        Page;
        Writeln('Enter a Deposit to ', Transaction.
            CheckOrSave);
        Writeln;
        GetAmount;
        Write('Enter Date MM/DD/YY: ');
        Readln(Date);
        Code := 1;
    end {with}
end; {EnterDeposit}
procedure EnterInterest;
begin
    Page;
    Writeln(' Enter Interest to ', Transaction.CheckOrSave);
    Writeln;
    with Transaction do
        begin
            GetAmount;
            Code := 3;
            Write('Enter Date MM/DD/YY: ');
            Readln(Date);
        end {with}
    end; {EnterInterest}
procedure EnterFee;
begin
    with Transaction do
        begin
            Page;
            Writeln('Enter a Fee');
            Writeln;
            GetAmount;
            Write('Enter Date MM/DD/YY: ');
            Readln(Date);
            Code := 4;
        end
    end; {EnterFee}
procedure EnterWithdrawal;
begin
    with Transaction do
        begin
            Page;
            Writeln('Enter a Withdrawal to ', CheckOrSave);

```

```

        Writeln;
        GetAmount;
        Code := 2;
        Write('Enter Date MM/DD/YY: ');
        Readln(Date)
    end {With}
end; {EnterWithdrawal}
procedure PrintCheckingReport;
begin
    PrinterMessage;
    Writeln(Out, 'Checking Report');
    Seek(TransFile, 0);
    while not (eof(TransFile)) do
        begin
            if TransFile^.CheckorSave = Checking then
                with TransFile^ do
                    begin
                        Write(Out, Date);
                    case code of
                        1 :
                            Write(Out, 'Deposit');
                        2 :
                            begin
                                Writeln(Out, 'Check number ', CheckNumber, ' ':
                                    10, 'Deductible ', TaxDeduct
                                Write(Out, 'Paid to:', PayTO)
                            end;
                        3 :
                            Write(Out, 'Interest ');
                        4 :
                            Write(Out, 'Fee');
                    end; {case}
                    Writeln(Out, Amount : 7 : 2);
                    end; {with}
                    Get(TransFile)
                end; {While}
                Writeln(Out);
                Writeln(Out, 'Balance ', Balance.CheckBal : 7 : 2)
            end; {PrintCheckingReport}
procedure PrintSavingReport;
begin
    PrinterMessage;

```

```

Writeln(Out, 'Savings Report');
Seek(TransFile, 0);
while not (eof(transfile)) do
begin
    if TransFile^.CheckorSave = Savings then
    with TransFile^ do
    begin
        Write(Out, Date);
        case Code of
            1 :
                Write(Out, 'Deposit');
            2 :
                Write(Out, 'Withdrawal');
            3 :
                Write(Out, 'Interest ');
        end; {case}
        Writeln(Out, Amount : 7 : 2);
    end; {with}
    Get(TransFile)
end; {While}
Writeln(Out);
Writeln(Out, 'Balance ', Balance.SavBal : 7 : 2)
end; {PrintSavingsReport}
procedure SavingsOptions;
begin
    case Option of
        1 :
            begin
                EnterDeposit;
                Balance.SavBal := Balance.SavBal + Transaction.
                    Amount;
                WriteTransaction
            end;
        2 :
            begin
                EnterWithdrawal;
                Balance.SavBal := Balance.SavBal—Transaction.
                    Amount;
                WriteTransaction
            end;
        3 :
            begin

```



```

        EnterInterest;
        Balance.SavBal := Balance.SavBal + Transaction.
            Amount;
        WriteTransaction
    end;
4 :
    PrintSavingReport;
end {case}
end; {SavingsOptions}
procedure CheckingOptions;
begin
    case Option of
        1 :
            begin
                EnterDeposit;
                Balance.CheckBal := Balance.CheckBal +
                    Transaction.Amount;
                WriteTransaction
            end;
        2 :
            begin
                EnterCheck;
                Balance.CheckBal := Balance.CheckBal—
                    Transaction.Amount;
                WriteTransaction
            end;
        3 :
            begin
                EnterInterest;
                Balance.CheckBal := Balance.CheckBal +
                    /Transaction.Amount;
                WriteTransaction
            end;
        4 :
            begin
                EnterFee;
                Balance.CheckBal := Balance.CheckBal—
                    Transaction.Amount;
                WriteTransaction
            end;
        5 :
            PrintCheckingReport;

```

```

    end {case}
end;{CheckingOptions}
begin
    Stop := False;
    FourSet := [1, 2, 3, 4];
    FiveSet := [1, 2, 3, 4, 5];
    Open(BalanceFile, 'Balance.Data');
    Rewrite(out, 'Printer:');
    if eof(BalanceFile) = True then
        begin
            Balance.SavBal := 0;
            Balance.CheckBal := 0
        end {Read Balance File}
    else
        Balance := BalanceFile^;
        {OpenTransaction File}
        Open(TransFile, 'Trans.data');
        Seek(TransFile, MaxInt); {move to last position in file}
    repeat {main driver}
        DisplayMainMenu;
        InitRec;
        case option of
            1 :
                begin
                    Transaction.CheckOrSave := Checking;
                    CheckingMenu;
                    CheckingOptions
                end;
            2 :
                begin
                    SavingsMenu;
                    Transaction.CheckOrSave := Savings;
                    SavingsOptions
                end;
            3 :
                ShowBalances;
            4 :
                Stop := True;
        end;{case}
    until Stop = True;
    Close(TransFile);
    Seek(BalanceFile, 0);

```

```
BalanceFile^ := Balance;  
Put(BalanceFile); {Replace Balance record in file}  
Close(BalanceFile);  
end. {program}
```

The CheckingAndSavings program provides a strong framework from which many features can be added by writing new procedures. Some of the additions that you might consider are:

1. Adapt the report procedures to print only transactions after a given date.
2. Add a field to the transaction record to note if a check has cleared and then add a procedure to reconcile the checking account.
3. Adapt the checking report to print out out tax deductible expenses.

# Appendices

- A Selected Exercise Answers / 268
- B Menu Summary / 270
- C Documenting a Program / 275
- D Sound and Music / 284
- E Differences Between MacPascal and UCSD Pascal / 289
- F MacPascal Reserved Words / 292
- G MacPascal Syntax Diagrams / 293
- H List of QuickDraw Routines / 308
- I List of Sane Functions and Procedures / 314
- J MacPascal Error Messages / 317
- K Bibliography / 323
- L The Macintosh Character Set / 325



# A Selected Exercise Answers

## Chapter 4 (page 74)

1. a. True b. False c. False d. False e. False
2. a. False b. False c. True d. False e. True f. True
4. a. 15 b. 724 c. 15 d. 1 e. 39 f. 0
5. a. 

|   |   |
|---|---|
| 1 | 1 |
| 1 | 2 |
| 2 | 1 |
| 2 | 2 |
| 3 | 1 |
| 3 | 2 |

 b. 

|   |
|---|
| 1 |
| 2 |
| 3 |
| 4 |
| 5 |
| 6 |
| 7 |
| 8 |
| 9 |

## Chapter 6 (page 101)

1. a. real b. real c. integer d. integer
3. a.  $(X * X) + Y$  b.  $(A * X) + B$
4. a. **type**  
BasketballPosition = (Guard,Forward,Center)

## Chapter 8 (page 149)

1. a. **var** Ints = **array**[2..20] **of** integer;  
     b. **var** Float = **array**[-4..50] **of** real;  
     c. **var** Bools = **array**[-100..201] **of** boolean;
2. a. 61      b. 30      c. 33      d. 6
5. **for** i := 1 **to** 5 **do**  
     **for** j := 1 **to** 4 **do**  
         B[i,j] := -5;

## Chapter 9 (page 180)

1. a. 5  
     b. 6  
         5  
         4  
         3
3. 78
4. 18
7. **type** MailingRec = **record**  
     Name = **string**[40];  
     Address = **string**[30];  
     City = **string**[10];  
     State = **string**[2];  
     Zip = **string**[5]  
   **end**; { MailingRec }
11. a. [1,3,5,6,7,9]    b. []    c. [1,3,5,7]    d. [1,2]  
     e. ['A','E','I','O','U']

# B Menu Summary

## File Menu

The **File** menu contains options to save programs to the disk, restore programs from the disk, and print programs.

### *New*

Opens the **Program** window to allow you to start entering a new program.

### *Open*

Displays a dialog box that allows you to recall a program that is already stored on the disk. A program residing on a different disk can be recalled by ejecting the current disk and inserting the new disk into the drive. A program is opened by double clicking on its name or by selecting the name and then clicking on the **Open** button.

### *Close*

Closes the **Program** window and erases its contents. If you haven't saved the program you will be given a chance to do so.

### *Save*

Saves your program on the disk under the name that it has been previously given ( the name that appears on the top of the program window). If a program is new (named Untitled), a dialog box first asks for a name.

### *Save As . . .*

Saves a new program with a name or saves an already named program with a new name. It also allows you to eject the disk in

the disk drive so that you can save the program on a different disk.

***Revert***

After making changes to a program, **Revert** allows you to return to the unchanged version. A dialog box double checks that you really want to **Revert**.

***Page Setup***

Let's you select the size paper you are going to use in the printer.

***Print . . .***

Displays a dialog box that allows you to print the text of your program.

***Quit***

Returns you to the Macintosh desktop. If you have not saved your program you will be given an opportunity to do so.

## **Edit Menu**

The **Edit** menu contains options that allow you to make changes to the text of your program. Selections in this menu are only available if the program, **Observe** or **Instant** windows are active.

***Cut***

Cuts any selected text out of your program and places it into the Clipboard. The Clipboard is a temporary storage area that allows you to pass information between programs.

***Copy***

Does the same thing as **Cut**, except it does not remove the selected text from the current program.

***Paste***

Inserts a copy of the Clipboard at the insertion point in your program or replaces the selected text with it.

***Clear***

Erases any selected text.

***Select All***

Automatically selects all text in the program window.



## Search Menu

The **Search** menu contains options to easily locate and change text in the program window.

### *Find*

Looks for the text indicated with the **What to Find** command. The search starts at the current insertion point or at the end of the currently selected text. If the target text is found it is selected.

### *Replace*

Replaces the currently selected text with the text indicated in the **What to Find** command.

### *Everywhere*

Performs a **Find** followed by a **Replace** throughout the program, changing all occurrences of the indicated text.

### *What to Find*

Displays a dialog box that lets you indicate what text will be looked for by the **Find** and **Everywhere** commands. It also lets you indicate what the replacement text will be for a **Replace** or **Everywhere** command.

## Run Menu

The **Run** menu allows you to select different options for executing your program.

### *Check*

Checks the program in the program window for proper Pascal syntax, but doesn't execute the program. You may want to do this occasionally as you enter a large program to expedite the debugging process. **Check** is automatically performed by any command that executes your program.

### *Reset*

Causes a halted program to return to the pre-execution state. The **Finger** and the value of all variables are destroyed. The **Text** and **Drawing** windows are cleared.

### *Go*

Executes your program from the start or resumes execution of your program from where it was halted. A program executed

with **Go** will continue to run until either a break point set with **Stops-In** is hit or the end of the program is reached.

### **Go-Go**

Similar to **Go** except that when a break point is reached the program only is halted briefly to update the **Observe** window and then continues executing.

### **Step**

Causes execution of your program to start or executes the next line of your halted program. The **Finger** is displayed next to the next line to be executed. Repeated use of **Step** can be used to watch the sequence in which program statements execute.

### **Step-Step**

A cross between **Step** and **Go**. The program steps through the program automatically but does it slow enough to watch the **Finger** move. After each instruction the **Observe** window (if it is present) is updated.

### **Stops-In/Stops-Out**

Allows you to set break points (little **Stop** signs) in your program or displays **Stops** previously set. When **Stops-In** is selected the menu choice changes to **Stops-Out**. Selecting it will cause the **Stops** to disappear from your program.

## Windows Menu

Choosing any menu option except **Type Size** causes the window of the same name to be opened (if it is closed) and become the active window.

### **Untitled**

Opens the **Program** window. This menu choice will actually contain the name of the program window. It is called **Untitled** if the program window has not been saved with a name.

### **Instant**

Causes the **Instant** window to become active. The **Instant** window can be used to enter and immediately execute any Pascal statement on an ad hoc basis. This is useful while debugging since it gives you an opportunity to change the value of a variable while the program is halted.

***Observe***

Causes the **Observe** window to become active. The **Observe** window can be used to watch the values of a variable or expression as a program executes.

***Text***

Causes the **Text** window to become the active window. The **Text** window is where output to the standard text file is displayed.

***Drawing***

Causes the **Drawing** window to become the active window. The **Drawing** window is where graphics drawn by the Quickdraw routines appear.

***Clipboard***

Causes the **Clipboard** to become the active window giving you a chance to view what it contains.

***Type Size***

Displays a dialog box that allows you to select the size of the characters displayed on the screen.

***Pause***

This menu only appears while a program is executing. Holding down the mouse button on **Pause** stops execution until the button is released. If you choose **Halt** from the menu, your program will halt until you start it again with one of the choices from the **Run** menu.

# C Documenting a Program

The following program is presented for the purpose of providing a model for documenting a program. This particular program is only an example and should not be taken as the only way to document a program. The amount and detail of documentation that you should use in your program will depend on your application, it's length, and complexity.

```
program GradeBook;
{ ***** }
{ *   Created 1/7/84      Programmer: Alan Zeldin   * }
{ *                                                     * }
{ *                                                     * }
{ * Functional Description:                             * }
{ * GradeBook maintains a file containing student     * }
{ * names and grades. Weighted averages are computed  * }
{ * when student information is displayed.  Students * }
{ * can be added or deleted from the file and student * }
{ * grade information can be modified.                 * }
{ *                                                     * }
{ * Implementation Description:                         * }
{ * Information is maintained in a file of records    * }
{ * that contain data about each student. To find any * }
{ * specific record, a sequential search of the file   * }
{ * is performed.                                       * }
{ *                                                     * }
```



```

{ * Implementation Restrictions: (Bugs) * }
{ * Only the first student in the file with a given * }
{ * name is accessible to the user. * }
{ * * }
{ ***** }
const
  FileName = 'STUDENTS'; { Name of file students are
                           stored in }
  FileSize = 50;
  NumberExams = 6;
  Empty = ""; { Null string in LastName marks free
               record }
type
  Grades = array[1..NumberExams] of integer;
  NameType = string[15];
  StudentType = record
    LastName : NameType;
    FirstName : NameType;
    Exam : Grades;
  end;
var
  Choice : Char;
{ Weight of each exam in average }
  Weight : array[1..NumberExams] of Real;
  Student : file of StudentType;
  First, Last : NameType; { Name of current Student }
  Location : Integer; { Current location in Student file }
  TextWindow : Rect; { Hold dimensions of text
                      window}

procedure InitScreen;
{ ***** }
{ * Set up the text window for program output. * }
{ ***** }
begin
  HideAll;
  SetRect(TextWindow, 40, 40, 400, 300);
  SetTextRect(TextWindow);
  ShowText
end;

```

**procedure Menu;**

```

{ ***** }
{ *   Display all the program's functions and prompts the   * }
{ *   user to choose the desired function. If an invalid    * }
{ *   choice is made the computer beeps and waits for a     * }
{ *   a valid choice                                         * }
{ ***** }
begin { Menu }
    Page(output);
    Writeln('1. Modify a Student');
    Writeln('2. Add a Student');
    Writeln('3. Delete a Student');
    Writeln('0. Quit');
    Writeln;
    Write('Please Choose > ');
    Read(Choice);
    while not (Choice in ['0'..'4']) do
        begin { Not a valid Choice }
            SysBeep(5);
            Read(Choice)
        end
    end; { Menu }

```

**procedure InitFile;**

```

{ ***** }
{ *   Open Student file if it exists else create the file   * }
{ ***** }
var
    Count, Index : integer;
begin { InitFile }
    Open(Student, FileName);
    if EOF(Student) then { file doesn't exist }
        begin
            { Write FileSize empty records into the file }
            for Count := 0 to FileSize do
                begin
                    Seek(Student, count);
                end
            { Initialize all records in file to be empty }
            with Student^ do

```

```

        begin
            FirstName := Empty;
            LastName := Empty;
            for Index := 1 to NumberExams do
                Exam[Index] := 0
            end;
            Put(Student)
        end;
        Close(Student);
        Open(Student, FileName)
    end
end; { InitFile }

procedure SetWeights;
begin
    Weight[1] := 0.15;      { 15% of Grade }
    Weight[2] := 0.15;      { 15% of Grade }
    Weight[3] := 0.20;      { 20% of Grade }
    Weight[4] := 0.15;      { 15% of Grade }
    Weight[5] := 0.15;      { 15% of Grade }
    Weight[6] := 0.20;      { 20% of Grade }
end; { SetWeights }

procedure GetName;
{ ***** }
{ *   Prompt user to enter name of a Student   * }
{ ***** }
begin { GetName }
    Write('Enter Last Name >');
    Readln>Last);
    Write('Enter First Name >');
    Readln(First)
end; { GetName }

function FindStudent (First, Last : NameType;
    var Count : Integer) : boolean;
{ ***** }
{ *   Returns true if the name is found in the file.   * }
{ *   Returns false if the name is not found. If the name   * }
{ *   was found , Count contains its position in the file   * }
{ ***** }

```

```

var
    Found : Boolean;
begin { FindStudent }
    Count := 0;
    Found := False;
    Seek(Student, Count);
{ Sequential Search of file for matching name }
    while (not eof(Student)) and (not Found) do
        begin
            with Student^ do
                if (FirstName = First) and (LastName = Last) then
                    Found := True
                else
                    begin
                        Count := Count + 1;
                        Seek(Student, Count)
                    end
            end;
        end;
    if not found then
        begin
            Writeln(First, ' ', Last, ' was not found. ');
            Writeln('Press <Return> to continue. ');
            Readln
        end;
    FindStudent := Found
end; { FindStudent }

```

```

procedure DisplayStudentInfo;
{ ***** }
{ *   Display the contents of the current student record   * }
{ *   which is contained in Student^. Calculate the average * }
{ ***** }
var
    Index : Integer;
    Average : Real;
begin { DisplayStudentInfo }
    with Student^ do
        begin
            Page(output);
            Writeln(' Last Name : ', LastName);
            Writeln(' First Name : ', FirstName);

```



```

        Writeln;
        Writeln('1. Exam1 : ', Exam[1]);
        Writeln('2. Exam2 : ', Exam[2]);
        Writeln('3. Final : ', Exam[3]);
        Writeln('4. Project1: ', Exam[4]);
        Writeln('5. Project2: ', Exam[5]);
        Writeln('6. Project3: ', Exam[6]);
        Average := 0;
        for Index := 1 to NumberExams do
            Average := Average + (Weight[Index] *
                Exam[Index]);
        Writeln('    Average : ', Round(Average))
    end
end; { DisplayStudentInfo }

procedure Modify (Location : Integer);
{ ***** }
{ *   Allows user to modify exam grades of the current   * }
{ *   student.                                           * }
{ ***** }
var
    Choice : Char;
    Value : Integer;
begin { Modify }
    Seek(Student, Location);
    repeat
        DisplayStudentInfo;
        Writeln;
        Write('Enter Line to Change or 0 to Quit > ');
        Read(Choice);
        while not (Choice in ['0'..'6']) do
            begin { Not a valid Choice }
                SysBeep(5);
                Read(Choice)
            end;
        Writeln;
        if Choice <> '0' then
            begin
                Write('Enter value > ');
                Readln(Value);
                { Convert choice into equiv integer for use as index }
                Student^.Exam[Ord(Choice) - Ord('0')] := Value;

```

```

    end;
    until Choice = '0';
    Put(Student)
end; { Modify }

```

```

procedure AddStudent (First, Last : NameType);
{ ***** }
{ *   Adds the Student whose name is passed in to the file   * }
{ ***** }
    var
        Added : Boolean;
        Count : Integer;
    begin { AddStudent }
    { Look for a record with no last name }
        Added := False;
        Count := 0;
        Seek(Student, Count);
    { Sequential Search of file for free record }
        while (not EOF(Student)) and (not Added) do
            if Student^.LastName = Empty then
                begin { found an empty spot }
                    Seek(Student, Count);
                    Student^.LastName := Last;
                    Student^.FirstName := First;
                    Put(Student);
                    Added := True
                end
            else
                begin { look at next spot }
                    Count := Count + 1;
                    Seek(Student, Count)
                end;
            if not Added then
                Writeln('File is full, Press <Return> to continue')
        end; { AddStudent }

```

```

procedure Delete (Location : integer);
{ ***** }
{ *   Deletes the student at the place in the file specified   * }
{ *   by Location by putting an empty string into that         * }
{ *   record. The user is asked to verify the deletion.         * }
{ ***** }

```

```
var
    Which : Char;
begin { Delete }
    Seek(Student, Location);
    with Student^ do
        Writeln(FirstName, ' ', LastName);
        Write(' Delete (Y/N) > ');
        Read(Which);
        if (Which = 'Y') or (Which = 'y') then
            begin
                Seek(Student, Location);
                Student^.LastName := Empty;
                Student^.FirstName := Empty;
                Put(Student);
            end
    end; { Delete }

begin { GradeBook }
    InitScreen;
    InitFile; { file doesnt already exists, create it }
    SetWeights;
    repeat
        Menu; { Show the choices }
        case Choice of
            '0' :
                ; { Do Nothing }
            '1' : { Modify a Student }
                begin
                    Page(output);
                    GetName;
                    if FindStudent(First, Last, Location) then
                        Modify(Location)
                    end;
                end;
            '2' : { Add a Student }
                begin
                    Page(output);
                    GetName;
                    AddStudent(First, Last)
                end;
        end;
```

```
'3' :      { Delete a Student }  
  begin  
    Page(output);  
    GetName;  
    if FindStudent(First, Last, Location) then  
      Delete(Location)  
    end  
  end { case }  
until Choice = '0';  
Close(Student)  
end. { GradeBook }
```



# D Sound and Music

MacPascal provides the ability to exploit the Macintosh's built in music synthesizer. The MacPascal's Note procedure is used to produce tones through the Macintosh's built-in speaker. The procedure can be used for producing a simple beep to alert the user of a program, or for producing complex musical passages. The form for the Note procedure is:

**Note(Frequency, Amplitude, Duration);**

Frequency is the frequency or pitch of the tone to be produced in Hertz (Cycles per second). Frequency is a longint type. Amplitude is an integer type that determines how loud the sound to be produced is. The value for Amplitude can range from 0, which is silent, to 255, the loudest volume the Macintosh can produce. Duration is an integer that specifies the length of time a Note should sound in 60<sup>ths</sup> of a second. The following statement will sound the Note middle C for 1 second.

**Note(256,200,60)**

Middle C has the frequency 256Hz and the Note sounds for 60 \* 1/60 or 1 second.

The rest of this appendix on sound goes into some technical aspects of music and may be omitted by those not interested in producing music.

Musicians usually don't think of a tone's pitch in terms of its frequency. Instead they think of a tone's pitch in terms of notes on a scale. Most music works with a system that divides each octave into twelve equal parts (equal temperament). Two tones that are an octave apart have the same fundamental quality but at a different pitch. If Note X is exactly one octave higher than

Note Y, then the frequency of Note X will be exactly double that of Note Y. Let's look at a simple program that will play the Note C in three octaves.

```

program OctaveDemo;
  var
    I : Integer;
    Pitch : Integer;
  begin
    Pitch := 256 { Middle C }
    for I := 1 to 3 do
      begin
        Note(Pitch,200,60);
        Pitch := Pitch * 2
      end
    end.

```

Let us denote middle C as C, the Note one octave above middle C as C1, the Note two octaves above middle C as C2, and so forth. Notice that the distance in terms of frequency from middle C to C1 is (512 - 256) or 256Hz. The distance in hertz from the C1 to C2 is (1024 - 512) or 512Hz. The distance between different octaves in hertz changes as the frequency gets higher. Therefore we can not break up an octave into equal pieces by simply dividing the octave by the number of pieces. If we break an octave up into one equally spaced tone, we simply multiply by 2 to get to the next octave. If we break an octave into 2 equally spaced tones, we would multiply by  $\sqrt{2}$  or  $2^{1/2}$  to get halfway to the next octave and by  $\sqrt{2}$  or  $2^{1/2}$  again to get all the way to the next octave. If we break up an octave into 3 equally spaced tones we would multiply by the cubed root of 2 or  $2^{1/3}$  to get one third of the way to the next octave and multiply again by  $2^{1/3}$  for each additional third of an octave. In a similar fashion, we could break an octave into twelve equally spaced tones by multiplying by  $2^{1/12}$  to get from each tone to the next.

To multiply by  $2^{1/12}$  we need to use the XpwrY function. This function has the form

XpwrY(X,Y)

This function returns the value  $X^Y$  where X, Y and the value returned are all of type Extended (a high precision version of the real data type). In order to use this function and other special arithmetic functions you will need to use the SANE unit. For

more information on the SANE (Standard Apple Numeric Environment) unit refer to Appendix D of the Macintosh Pascal Technical Appendix manual). To use the SANE unit use the following statement after the program statement:

```
uses  
    SANE;
```

The program Tones12 will produce all twelve tones for the octave starting at middle C. Notice that the frequency of the tone is stored in a real variable and rounded only when it is used to minimize rounding errors.

```
program Tones12;  
    uses  
        SANE;  
    var  
        Tone : Real;  
        Counter : Integer;  
    begin  
        Tone := 256; { pitch of middle C }  
        for counter := 1 to 13 do  
            begin  
                Note(Round(Tone),200,30);  
                Tone := Round(Rone * XpwrY(2,1/12)) { calculate  
                    next tone }  
            end  
        end.
```

The following program illustrates how you can use the Note procedure to produce actual music. In this program we use a two dimensional array to store the pitch for each tone in the scale. The pitch values for every tone are calculated and stored in an array before the music begins playing in order to avoid the delay of calculating each new Note as the music is playing. This program uses constants and the user defined type NoteType to make entering the pitch and durations easy and meaningful.

```
program SoundDemo;  
    uses  
        SANE;  
    const  
        EN = 6;      { Eighth Note }  
        QN = 12;     { Quarter Note }
```

```

HN = 24;      { Half Note }
WN = 48;      { Whole Note }
DEN = 9;      { Dotted Eighth Note }
DQN = 18;     { Dotted Quarter Note }
DHN = 36;     { Dotted Half Note }
DWN = 72;     { Dotted Whole Note }

```

**type**

```

NoteType = (A, Bb, B, C, Cx, D, Eb, E, F, Fx, G, Ab);
OctaveType = 1..5;
Frequency = LongInt;

```

**var**

```

Notes : array[0..11, OctaveType] of LongInt;
NoteCounter : Integer; { NoteType }
OctaveCounter : OctaveType;
SeedTone, Next : Real;

```

**begin**

```

Next := XpwrY(2, 1 / 12);    { twelv'th root of 2 }
SeedTone := 110.0;           { 3rd A below middle C }
    { Set up an array containing all the tones frequency }
for OctaveCounter := 1 to 5 do
for NoteCounter := Ord(A) to Ord(Ab) do
    begin
        Notes[NoteCounter][OctaveCounter] := Round(SeedTone);
        SeedTone := SeedTone * Next { calculate next frequency }
    end;

```

{ play the tune Beethoven's Ninth }

```

Note(Notes[Ord(A), 3], 150, QN);
Note(Notes[Ord(A), 3], 150, QN);
Note(Notes[Ord(Bb), 3], 150, QN);
Note(Notes[Ord(C), 3], 150, QN);
Note(Notes[Ord(C), 3], 150, QN);
Note(Notes[Ord(Bb), 3], 150, QN);
Note(Notes[Ord(A), 3], 150, QN);
Note(Notes[Ord(G), 2], 150, QN);
Note(Notes[Ord(F), 2], 150, QN);
Note(Notes[Ord(F), 2], 150, QN);
Note(Notes[Ord(G), 2], 150, QN);
Note(Notes[Ord(A), 3], 150, QN);
Note(Notes[Ord(A), 3], 150, DQN);
Note(Notes[Ord(G), 2], 150, DEN);
Note(Notes[Ord(G), 2], 150, HN);

```



```
Note(Notes[Ord(A), 3], 150, QN);
Note(Notes[Ord(A), 3], 150, QN);
Note(Notes[Ord(Bb), 3], 150, QN);
Note(Notes[Ord(C), 3], 150, QN);
Note(Notes[Ord(C), 3], 150, QN);
Note(Notes[Ord(Bb), 3], 150, QN);
Note(Notes[Ord(A), 3], 150, QN);
Note(Notes[Ord(G), 2], 150, QN);
Note(Notes[Ord(F), 2], 150, QN);
Note(Notes[Ord(F), 2], 150, QN);
Note(Notes[Ord(G), 2], 150, QN);
Note(Notes[Ord(A), 3], 150, QN);
Note(Notes[Ord(G), 2], 150, DQN);
Note(Notes[Ord(F), 2], 150, DEN);
Note(Notes[Ord(F), 2], 150, HN);
end.
```

Enter and run the SoundDemo program to hear Beethoven's Ninth Symphony. To play other melodies, simply replace the parameters to the Note procedure with those for your melody.

There are many possibilities for the enhancement of the SoundDemo program. You could read in a tune into an array from a file containing pitch, amplitude, and duration information. You could create a program to edit such a file using QuickDraw graphics to represent standard musical notation on the musical staff. MacPascal provides the capabilities, you must supply the imagination.

# E Differences Between MacPascal and UCSD Pascal

A programmer experienced in using UCSD Pascal will find little difficulty in programming with MacPascal. MacPascal has many similarities with UCSD Pascal and only a few minor differences. This Appendix will summarize them.

## Data Types

The UCSD and MacPascal type LONGINT are not the same. The UCSD LONGINT is a BCD representation of an integer for decimal arithmetic and is similar to the MacPascal real type COMPUTATIONAL. UCSD has no equivalent to the MacPascal real types DOUBLE and EXTENDED.

## Strings

String handling is virtually the same in both systems. The MacPascal default string size is 255 rather than 80. All the standard functions and procedures are the same. MacPascal has added two procedures, Include and Omit. MacPascal does not support turning off range checking nor any other UCSD compiler switch.

## Case Statement

MacPascal has an Otherwise clause in the Case statement. UCSD Pascal does not.

## User-Defined Types

MacPascal allows the reading and writing of the values of a user-defined type. UCSD Pascal permits only the Ords of user-defined values to be read or written.

## Files

There are significant differences in file handling between the two systems. MacPascal tries to implement file handling more consistent with the original Pascal definition. In MacPascal there is a sharp distinction between sequential and random files, in UCSD Pascal there is not. A MacPascal sequential file can be open only to read or write. No mixed operations are allowed. To read then write to a sequential file the file has to be opened for reading, closed, and then reopened for writing. Here is a run-down on the differences in the file handling procedures.

**Reset** Used in MacPascal to open a sequential file for read only

**Rewrite** Used in MacPascal to open a sequential file for write only.

**Open** Used in MacPascal to open a random file.

**Close** Same as in UCSD Pascal

**Seek** Seek, in MacPascal, can only be used with a file opened for random access. Seek always performs a Get.

**Get** Same in both systems.

**Put** Same in both systems.

## Miscellaneous

In UCSD Pascal a carriage return sets EOLN to be true but returns the same ASCII code as a space (32). In MacPascal the ASCII code returned is 13. MacPascal automatically formats a program as it is entered, UCSD doesn't.

## Implementation

UCSD Pascal utilizes an editor, compiler, linker, and interpreter. A program is first edited, it is then compiled, and linked into an intermediate form known as P-code. The P-code is run with the use of a P-code interpreter. MacPascal is a totally interactive system reminiscent of interpreted BASIC. All program editing and execution is performed by the MacPascal interpreter.



# F MacPascal Reserved Words

## Reserved Words

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

## Single Character Special Symbols

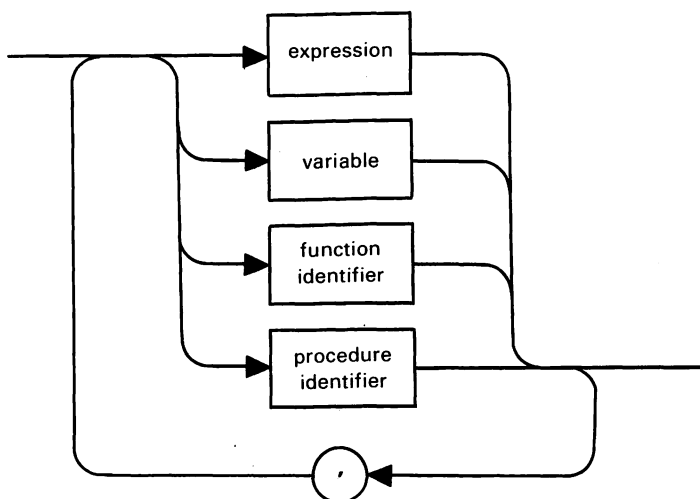
`$ ( ) * + , - . / : ; < = > @ [ ] ^ { }`

## Character Pair Special Symbols

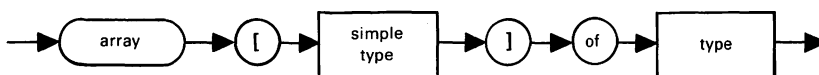
`< > < = > = : = .. ( * ) ( . )`

# G MacPascal Syntax Diagrams

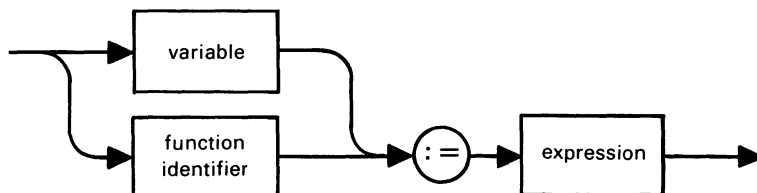
ARGUMENT LIST



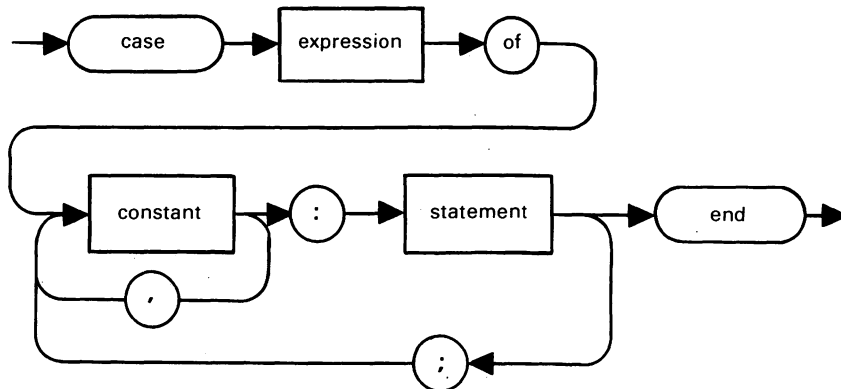
ARRAY TYPE



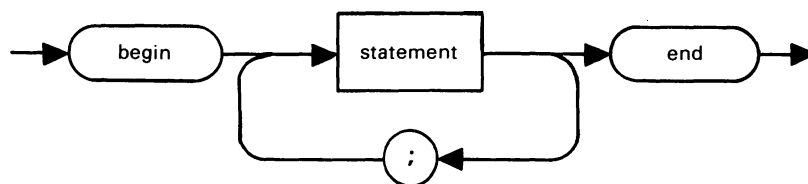
### ASSIGNMENT STATEMENT

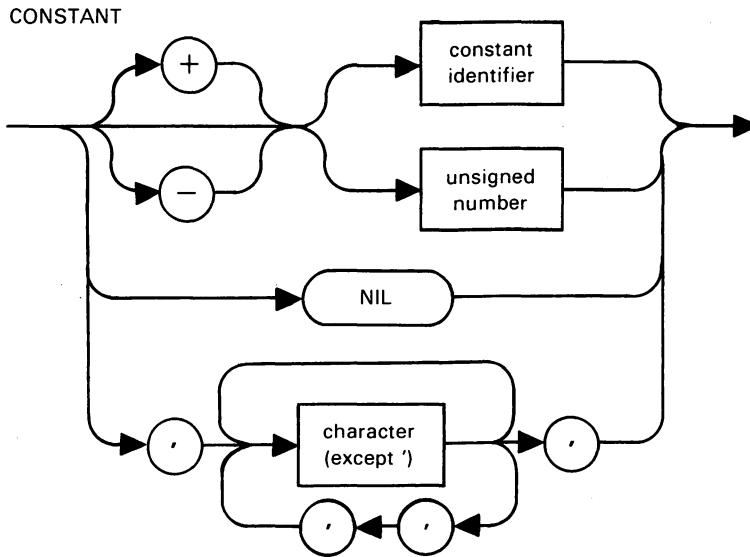


### CASE STATEMENT

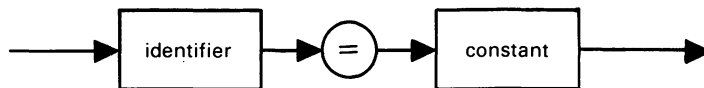


### COMPOUND STATEMENT



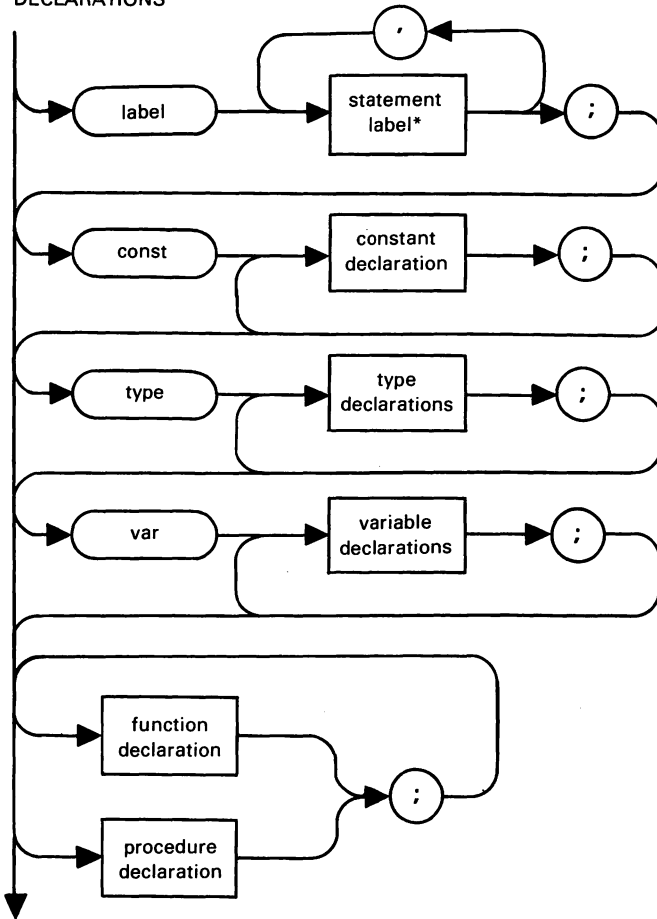


CONSTANT DECLARATION



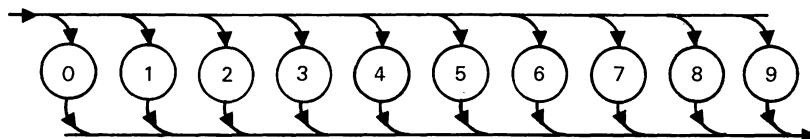


# DECLARATIONS

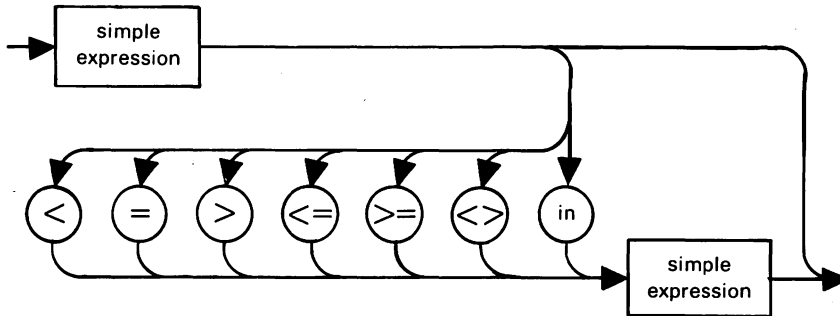


\*statement label is one-to-four-digit unsigned integer

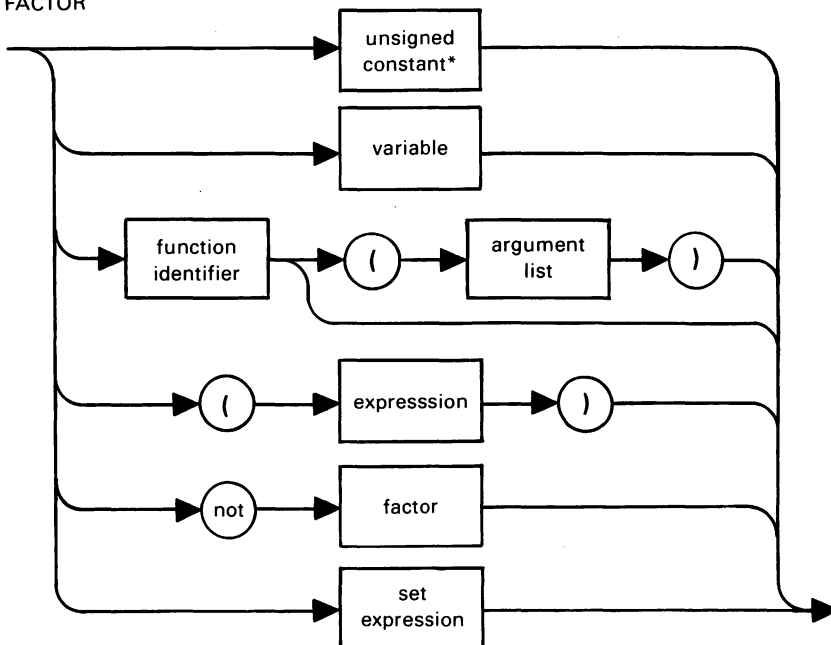
# DIGIT



EXPRESSION

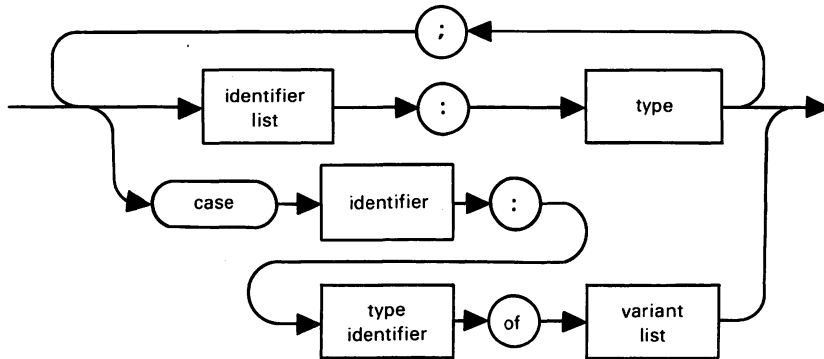


FACTOR

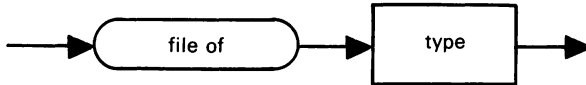


\*An unsigned constant is a constant without a leading sign.

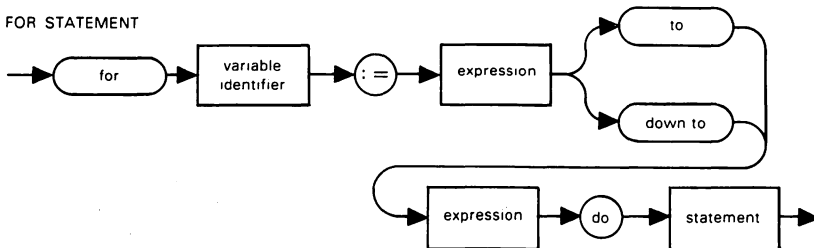
# FIELD LIST



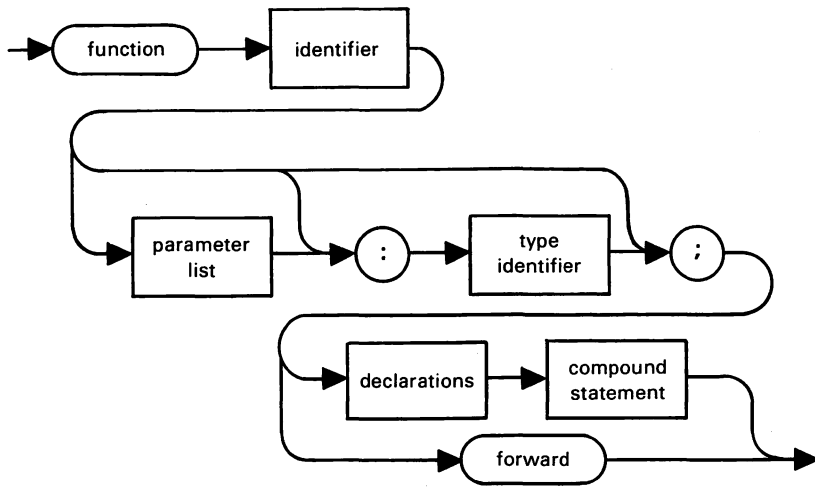
# FILE TYPE



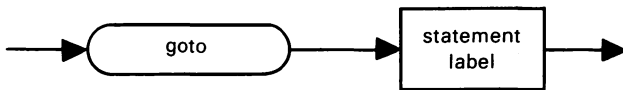
# FOR STATEMENT



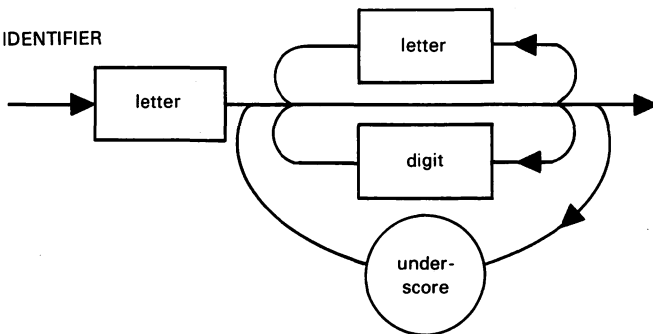
# FUNCTION DECLARATION



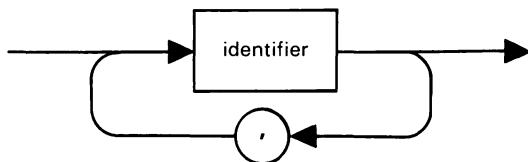
# GOTO STATEMENT



# IDENTIFIER

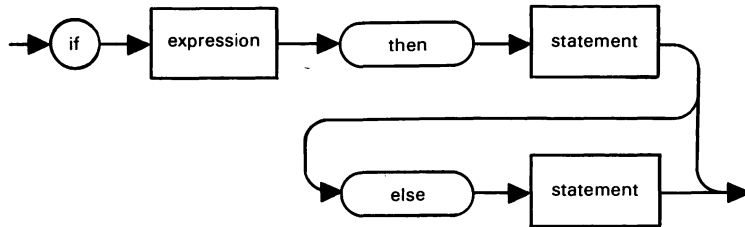


# IDENTIFIER LIST

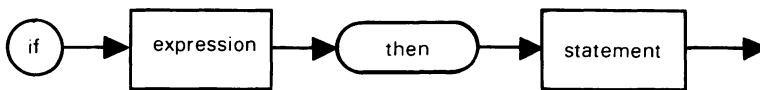




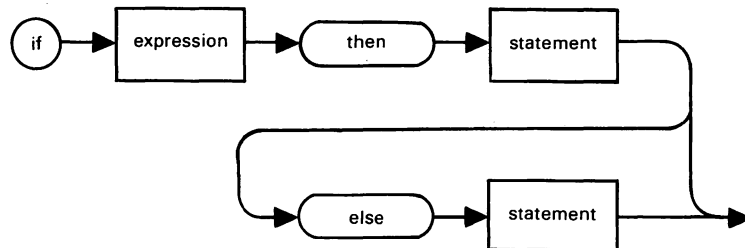
IF STATEMENT



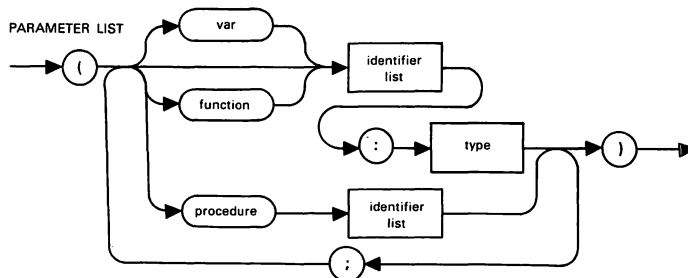
IF THEN STATEMENT



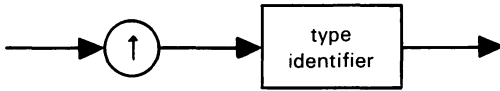
IF THEN ELSE STATEMENT



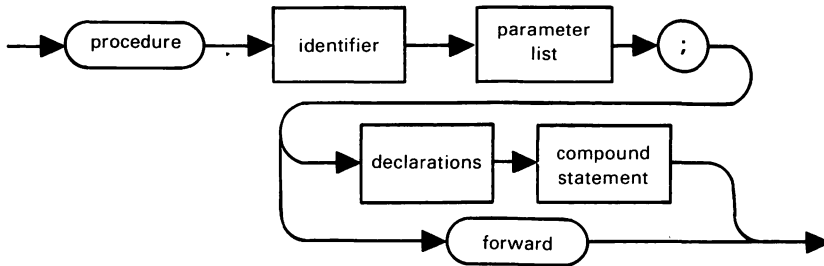
PARAMETER LIST



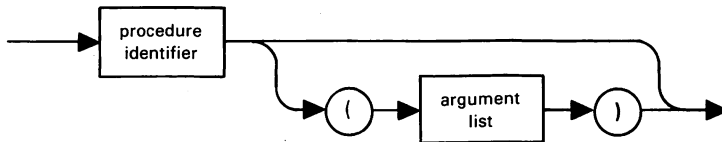
POINTER TYPE



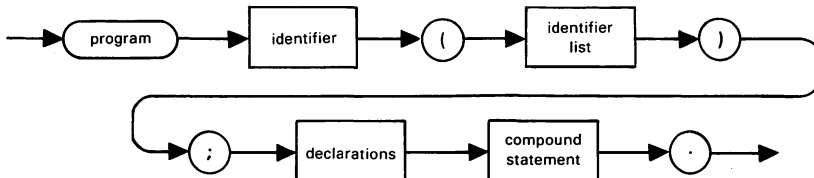
PROCEDURE DECLARATION



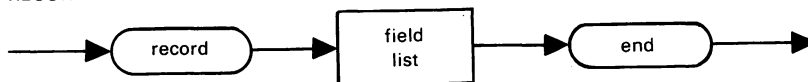
PROCEDURE STATEMENT



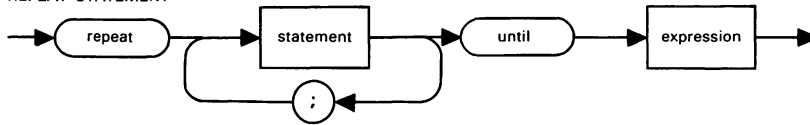
PROGRAM



RECORD TYPE



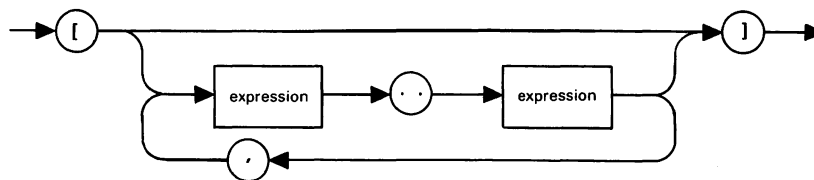
REPEAT STATEMENT



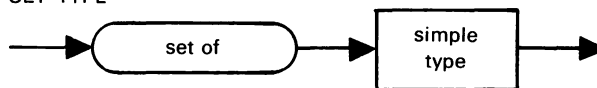
SCALAR TYPE



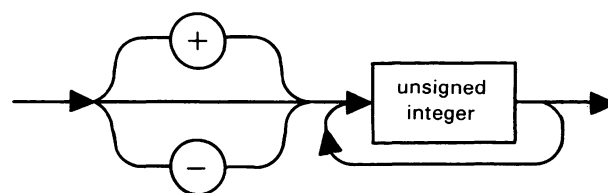
SET EXPRESSION



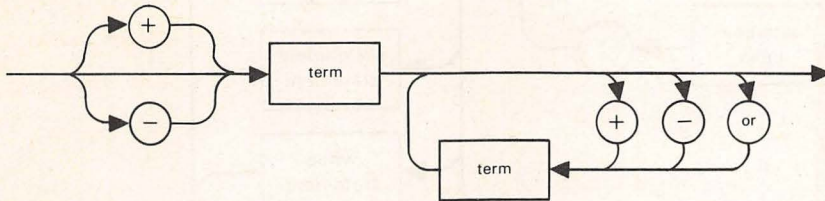
SET TYPE



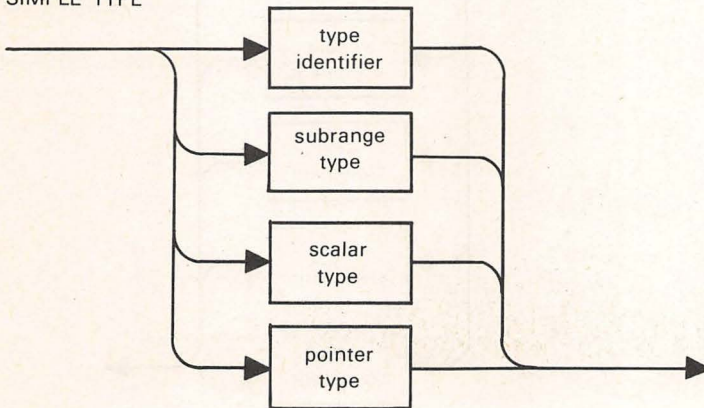
SIGNED INTEGER



SIMPLE EXPRESSION

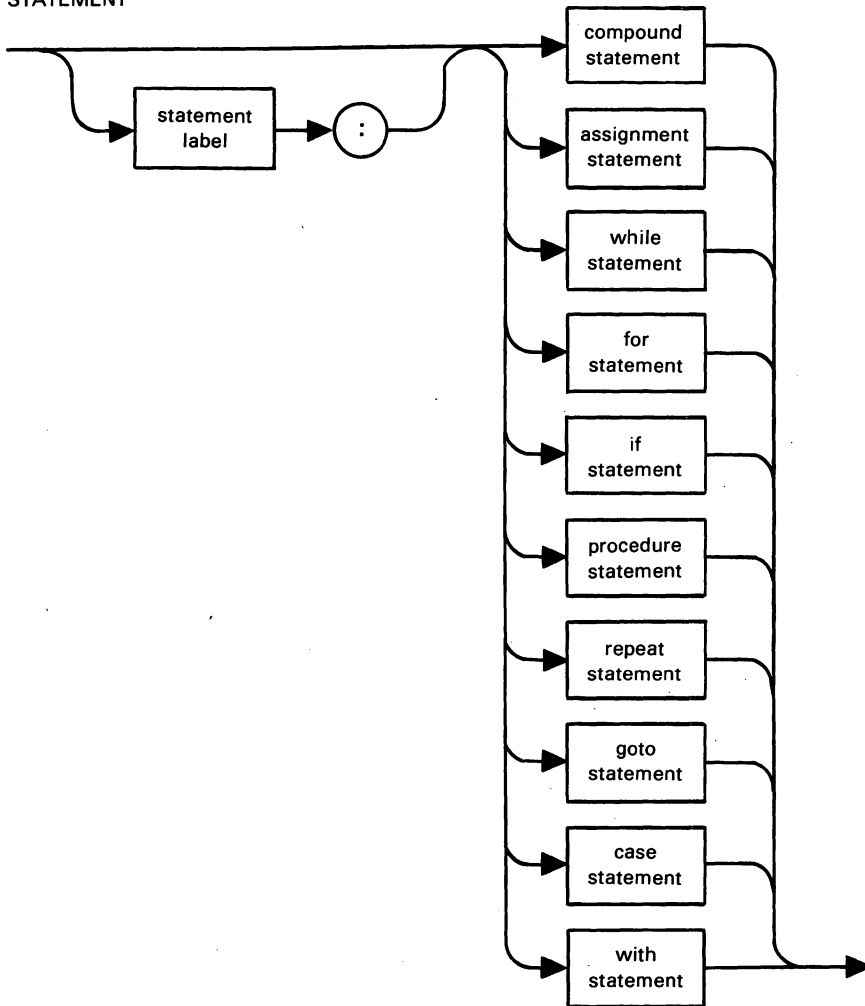


SIMPLE TYPE



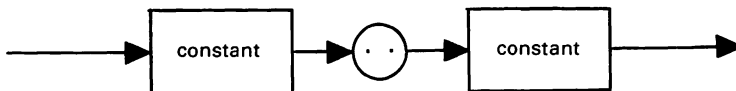


STATEMENT

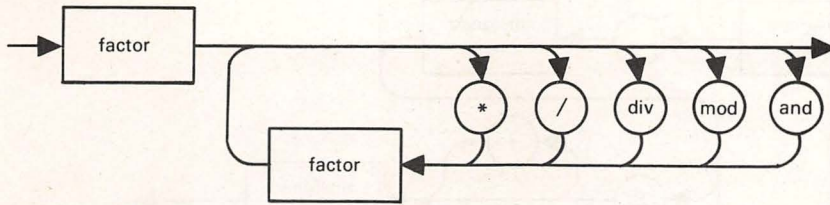


STATEMENT LABEL      one-to-four-digit unsigned integer

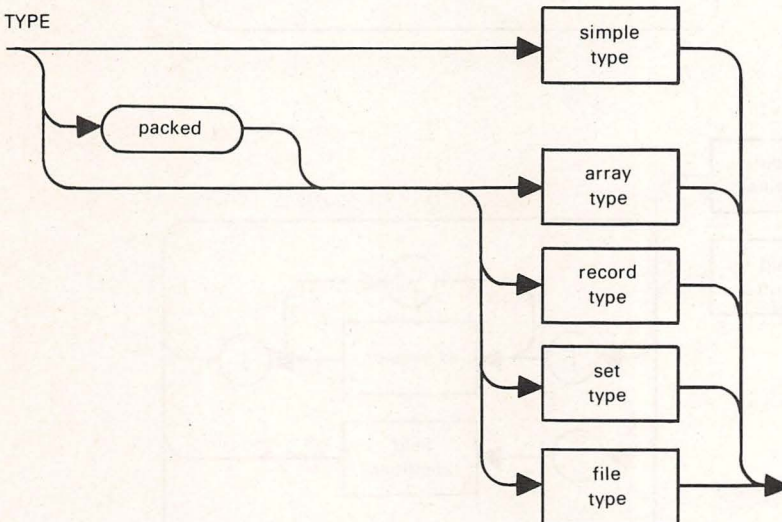
SUBRANGE TYPE



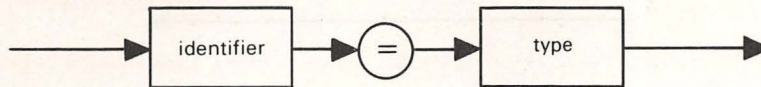
TERM



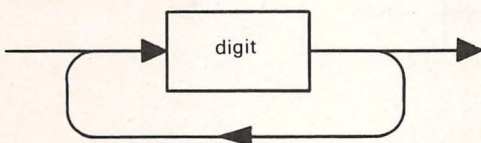
TYPE



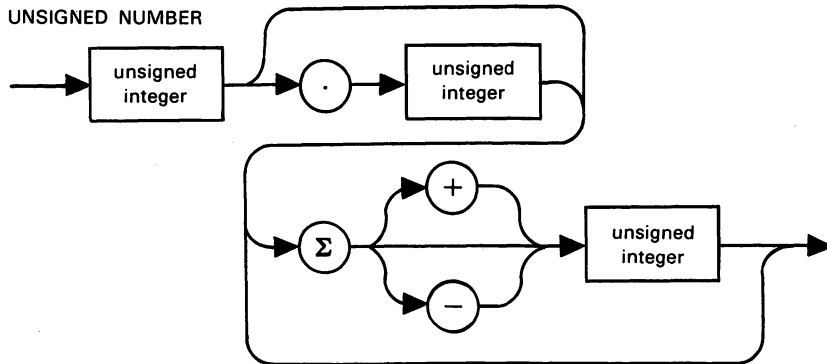
TYPE DECLARATION



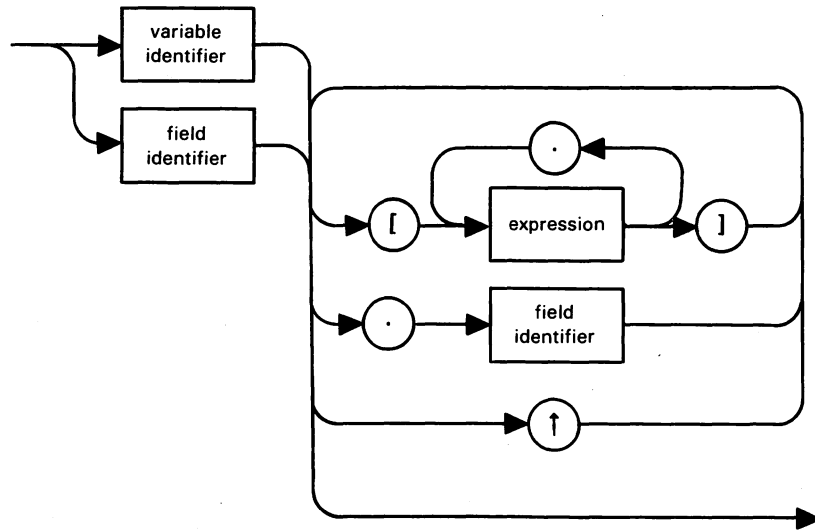
UNSIGNED INTEGER



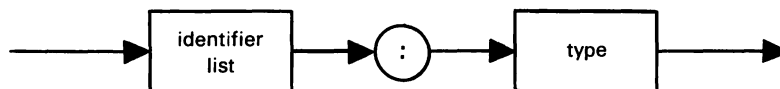
UNSIGNED NUMBER



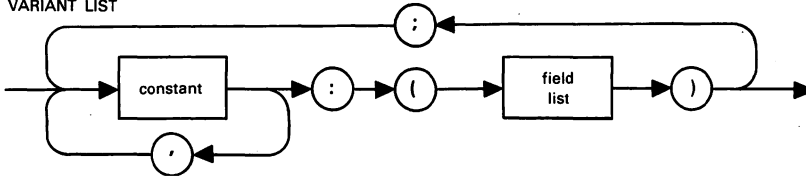
VARIABLE



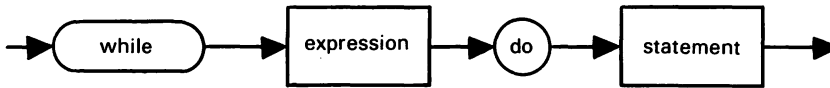
VARIABLE DECLARATION



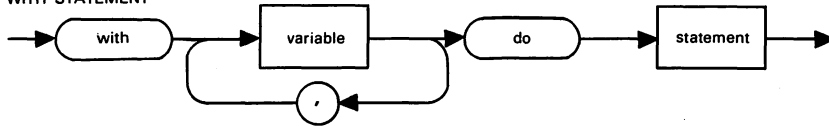
VARIANT LIST



WHILE STATEMENT



WITH STATEMENT





# H List of QuickDraw Routines

This appendix lists the routines contained in the QuickDraw1 and QuickDraw2 units. The numbers that appear in brackets indicate which QuickDraw unit the routine is part of.

## 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);  
{1} 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);
```

## Cursor Routines

```
{1} procedure InitCursor;  
{1} procedure SetCursor (Crsr: Cursor);
```

```

{1} procedure HideCursor;
{1} procedure ShowCursor;
{1} procedure ObscureCursor;

```

## 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, Vv : 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 TextFace (Face : Style);
{1} procedure TextMode (Mode : Integer);
{1} procedure TextSize (Size : Integer);
{1} procedure SpaceExtra (Extra : LongInt);
{1} procedure DrawChaw (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);
{1} procedure GetFontInfo (var Info : FontInfo);

```

## Point Calculations

```

{1} procedure AddPt (sSrc : 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; FromRect, ToRect :  
    Rect);  
{1} procedure MapPt (var Pt : Point; FromRect, ToRect :  
    Rect);  
{1} procedure LocalToGlobal (var Pt : Point);  
{1} procedure GlobalToLocal (var Pt : Point);
```

## Rectangle Routines

```
{1} procedure FrameRect (R : Rect);  
{1} procedure PaintRect (R : Rect);  
{1} procedure EraseRect (R : Rect);  
{1} procedure InvertRect (R : Rect);  
{1} procedure FillRect (R : Rect; Pat : Pattern);
```

## 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);
```

## 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);
```

## Arc Routines

```
{1} procedure FrameArc (R : Rect; StartAngle, ArcAngle :  
    Integer);
```



```

{1} procedure PaintArc (R : Rect; StartAngle, ArcAngle :
      Integer);
{1} procedure EraseArc (R : Rect; StartAngle, ArcAngle :
      Integer);
{1} procedure InvertArc (R : Rect; StartAngle, ArcAngle :
      Integer);
{1} procedure FillArc (R : Rect; StartAngle, ArcAngle :
      Integer; Pat : Pattern);
{1} procedure PtToAngle (R : Rect; Pt : Point; var Angle :
      Integer);

```

## 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);

```

## Region Calculations

```

{2} function NewRgn : RgnHandle;
{2} procedure DisposeRgn (Rgn : RgnHandle);
{2} procedure CopyRgn (SrcRgn, dstRgn : RgnHandle);
{2} procedure SetEmptyRgn (Rgn : RgnHandle);
{2} procedure SetRectRgn (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 Regions

```
{2} procedure FrameRgn (Rgn : RgnHandle);  
{2} procedure PaintRgn (Rgn : RgnHandle);  
{2} procedure EraseRgn (Rgn : RgnHandle);  
{2} procedure InvertRgn (Rgn : RgnHandle);  
{2} procedure FillRgn (Rgn : RgnHandle; Pat : Pattern);
```

## Graphical Operations on Bitmaps

```
{2} procedure ScrollRect(DstRect : Rect; Dh, Dv : Integer;  
    UpdateRgn : RgnHandle);  
{2} procedure CopyBits (SrcBits, DstBits : BitMap;  
    SrcRect, DstRect : Rect;  
    Mode : Integer;  
    MaskRgn : RgnHandle);
```

## 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);

```

## The Bottleneck Interface

```

{2} procedure SetStdProcs (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 StdRRect( 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 Number, Denom : Point;
    var Info : FontInfo) : Integer;
{2} procedure StdGetPic (DataPtr :QDPtr; ByteCount :
    Integer);
{2} procedure StdGetPic (DataPtr :QDPtr; ByteCount :
    Integer);

```

## Miscellaneous Utility Routine

```

{1} function GetPixel (H, V : Integer) : Boolean;
{1} function Random : Integer;
{1} procedure StuffHex (ThingPtr : QDPtr; S : Str255);
{2} procedure ForeColor (Color : LongInt);
{2} procedure BackColor (Color : LongInt);
{2} procedure ColorBit (WhichBit : Integer);

```

# List of Sane Functions and Procedures

This Appendix provides a reference to the Sane unit. This unit implements the Institute of Electrical and Electronics Engineers (IEEE) Standard 754 for Binary Floating-Point Arithmetic. To use any of the following procedures or functions in your program, simply include the following statement immediately following the program declaration statement:

```
uses  
    Sane;
```

For more detailed information on the Sane library consult Appendix D of the Macintosh Pascal Technical Appendix.

## Transfer Routines

```
function Num2Integer( X : Extended ) : Integer;
```

```
function Num2Longint( X : Extended ) : longInt;
```

```
procedure Num2Dec( F : DecForm; X : Extended; var D :  
    Decimal );
```

```
function Dec2Num( D : Decimal ) : Extended;
```

```
procedure Num2Str( F : DecForm; X : Extended; var S :  
    string );
```

```
function Str2Num( S : string ) : Extended;
```



## Comparison Routine

**function** Relation( X, Y : Extended ) : RelOp;

## Arithmetic, Auxiliary, and Elementary Function Routines

**function** Remainder( X, Y : Extended; **var** I : Integer ) :  
Extended;

**function** Rint( X : Extended ) : Extended;

**function** Scalb( N : Integer; X : Extended ) : Extended;

**function** Logb( X : Extended ) : Extended;

**function** CopySign ( X, Y : Extended) : Extended;

**function** NextReal( X, Y : Real ) : Real;

**function** NextDouble( X, Y : Double ) : Double;

**function** NextExtended( X, Y : Extended ) : Extended;

**function** Log2( X : Extended ) : Extended;

**function** Ln1( X : Extended ) : Extended;

**function** Exp2( X : Extended ) : Extended;

**function** Exp1(X : Extended ) : Extended;

**function** XpwrI( X : Extended; I : Integer ) : Extended;

**function** XpwrY( X, Y : Extended ) : Extended;

**function** Compound ( R, N : Extended ) : Extended;

**function** Annuity( R, N : Extended ) : Extended;

**function** Tan( X : Extended ) : Extended;

**function** Random( **var** X : Extended ) : Extended;



## Inquiry Routines

```
function ClassReal( X : Real ) : Numclass;  
function ClassDouble( X : Double ) : Numclass;  
function ClassComp( X : Comp ) : Numclass;  
function ClassExtended( X : Extended ) : Numclass;  
function SignNum( X : Extended ) : Integer;
```

## Environment Access Routines

```
procedure SetException( E : Exception; B : Boolean );  
function TestException( E : Exception ) : Boolean;  
procedure SetHalt( E : Exception; B : Boolean );  
function TestHalt( E : Exception ) : Boolean;  
procedure SetRound( R : RoundDir );  
function GetRound : RoundDir;  
procedure SetPrecision( P : RoundPre );  
procedure GetPrecision : RoundPre;  
procedure SetEnvironment( E : Environment );  
procedure GetEnvironment( var E : Environment );  
procedure SetEnvironRec( E : EnvironRec );  
procedure GetEnvironRec( var E : EnvironRec );  
procedure ProcEntry( var E : Environment );  
procedure ProcExit( E : Environment );
```

# J MacPascal Error Messages

This Appendix contains a listing of the possible syntax and Run Time Errors that might appear while running a MacPascal program.

## Syntax Errors

1. This doesn't make sense as a statement.
2. The name <name> has already been defined at this level.
3. An invalid variable, field or formal parameter list definition has been found. A colon might be missing.
4. The name <name> has not yet been defined.
5. A type or procedure name has been found where a variable, field name, or value is required.
6. A type is expected. <name> is defined but not as a type.
7. A constant is expected. <name> is defined but not as a constant.
8. A subrange boundary has been found whose type is not integer, char or enumerated.
9. A subrange has been found whose boundaries are not of the same type.
10. A subrange has been found whose lower boundary is greater than its upper boundary.
11. An array index has been found whose type is not integer, char, enumerated, or subrange.
12. An invalid enumerator list had been found.
13. A semicolon is required on this line or above but one has not been found.

14. Did not find a valid result type in the heading of the function's definition.
15. A colon is required on this line or above but one has not been found.
16. Either a semicolon or an **until** is expected following the previous statement, but neither has been found.
17. Either a semicolon or an **end** is expected following the previous statement, but neither has been found.
18. An invalid **program** parameter has been found.
19. **uses** can only appear immediately following the **program** heading.
20. A variable of **function** name is expected. <name> is defined, but not as a variable or a **function**.
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.
23. An invalid formal parameter list has been found.
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 field name of this **record**.
26. A **record** name is expected. name> is defined, but not as a record.
27. A **case** constant is required here but one has not been found.
28. An invalid variant definition has been found.
29. The size of this **string** should be a number between 1 and 255, but is not.
30. A **set** should have elements whose type is integer, char, or enumerated but this **set** does not.
31. The name <name> doesn't make sense here.
32. This label has not been defined.
33. A **program** key word was not found at the beginning of this program.
34. This statement or key word doesn't belong here.
35. This kind of declaration doesn't belong here.
36. At least one constant declaration is required after the key word **const**, but none has been found.
37. At least one variable declaration is required after the key word **var**, but none has been found.



38. At least one type declaration is required after the key word **type**, but none has been found.
39. **end.** is required at the end of a program, but was not found.
40. At least one library name is required after the key word **uses**, but none has been found.
41. An invalid library name has been found.
42. This is not allowed in the **Instant** window.
43. The value of this constant is not numeric and may not have a sign.
44. This does not make sense as a statement.
45. The available memory for variables defined at this level has been exhausted.
46. This declaration or statement does not belong here.
47. A variable of this type would be too large.
48. Too many up-arrows are being applied to <name>.
49. Too many indicies are being applied to <name>.
50. This attempt to assign a result to the **function** name <name> outside of its definition is invalid.
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.
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**.

## Run Time Errors

1. Excessive nesting or recursion of functions or procedures has occurred.
2. An incompatibility between types has been found.
3. Too many parameters have been used in a call to a procedure or function.
4. A constant or expression has been used as a parameter where a reference to a variable is required.
5. Too few parameters have been used in a call to a procedure or function.



6. The value of a variable or sub-expression is out of range for its intended use.
7. An attempt to perform an integer division by zero has occurred.
8. There is not enough memory to continue running.
9. **if**, **while**, and **until** require an expression that evaluates to **TRUE** or **FALSE**. This expression does not.
10. The value of an expression in a **case** statement above does not match its **case** constants.
11. An invalid **string** or **array** index has been found.
12. An attempt has been made to access a field in an invalid variant.
13. An attempt has been made to dereference a pointer whose value is **nil**.
14. A **string** value is too long for its intended use.
15. A value or expression in a set constructor is not **char**, **integer**, or **enumerated**.
16. There is a type mismatch between the lower and upper boundaries of a set operator.
17. You can't get there from here.
18. The **packed array** parameter for **Pack** or **UnPack** is not a **packed array**.
19. The **non-packed array** parameter for **Pack** or **UnPack** is not a **non-packed array**.
20. The index for **Pack** or **UnPack** is not a valid index for the **non-packed array**.
21. An attempt has been made to **goto** out of this level in which **Instant** or **Observe** is running.
22. The index for **Pack** or **UnPack** would cause too many elements to be transferred.
23. An attempt has been made to access data beyond the end of **file** or **string**.
24. An attempt has been made to access a **file** that has not been opened with **Rewrite**, **Reset**, or **Open**.
25. An attempt has been made to read from a **file** opened with **Reset**.
26. An attempt has been made to use a **file** of other files.
27. An attempt has been made to open a file already opened with **Rewrite**, **Reset**, or **Open**.
28. An attempt has been made to **Seek** with a file not opened with **Open**.
29. An invalid attempt to **Close** a file occurred.

30. An attempt has been made to use EOLN with a file that is not a TEXT file.
31. An attempt has been made to write to a file opened with Reset.
32. An attempt has been made to use a file buffer whose value is undefined.
33. An attempt has been made to use Readln or Writeln on a file that is not a TEXT file.
34. An attempt has been made to Dispose a pointer whose value was not created with New.
35. An invalid name was found while attempting to read an enumerated value.
36. Did not find TRUE or FALSE while attempting to read a Boolean value.
37. A colon modifier is invalid for the parameter of this **procedure** or **function**.
38. An attempt has been made to use a colon modifier whose value is less than one.
39. Floating point arithmetic exception: Invalid operation attempted.
40. Floating point arithmetic exception: Underflow occurred.
41. Floating point arithmetic exception: Overflow occurred.
42. Floating point arithmetic exception: Division by zero attempted.
43. An attempt has been made to use Page on a file that is not a TEXT file.
44. An attempt has been made to open a non-TEXT file to 'Printer:' or 'Modem:'.
45. An attempt has been made to Reset a file that does not exist.
46. An attempt has been made to use a modulus less than one with the **mod** operator.
47. The value of the control variable has been changed illegally while the **for** loop above was executed.
48. An attempt has been made to use the operator "<" or ">" on set values.
49. There is not enough memory to perform this **NEW** and more cannot be allocated while using **Instant**.
50. An attempt has been made to pass a **file** variable as a parameter that has not been declared **var**.
51. An error has occurred in a **procedure** or **function** in the program while using **Instant**.



- 52. An attempt has been made to read non-numeric data into a numeric variable.
- 53. An attempt has been made to use a parameter list with a name that is not a **procedure** or **function**.
- 54. An attempt has been made to access a file on disk which is not known.
- 55. An attempt has been made to access a file on a disk or volume which is not known to the system.
- 56. An attempt has been made to open a file that cannot be found.
- 57. An attempt has been made to write to a volume locked by a hardware setting.
- 58. An attempt has been made to write to a locked file.
- 59. An attempt has been made to write to a locked volume.
- 60. An attempt has been made to open a file which is already in use.
- 61. An attempt has been made to create a file with the same name as a file that already exists.

# K Bibliography

Aho, A., and Hopcroft, J., and Ullman, J., *Data Structures and Algorithms*, Addison Wesley, 1983.

*American National Standard Pascal Computer Programming Language*, ANSI/IEEE770X3.97-1983, IEEE/Wiley-Interscience, 1983.

*Inside Macintosh*, Apple Computer Inc., 1984.

Koffman, B., *Pascal a Problem Solving Approach*, Addison Wesley, 1982.

*Macintosh Pascal User's Guide*, Apple Computer Inc., THINK Technologies Inc., 1984.

*Macintosh Pascal Reference Manual*, Apple Computer Inc., THINK Technologies Inc., 1984.

*Macintosh Pascal Technical Appendix*, Apple Computer Inc., THINK Technologies Inc., 1984.

Cooper, D., *Standard Pascal User Reference Manual*, W. W. Norton & Co., 1983.

Gear, C., *Programming in Pascal*, Science Research Associates. Inc., 1983.

Hamacher, C. and Zvonko, V. and Safwat, Z., *Computer Organization*, McGraw Hill, 1978.

Horowitz, E. and Sahni, S., *Fundamentals of Data Structures*, Computer Science Press Inc., 1976.

Jensen, K. and Wirth, N., *Pascal User Manual and Report*, Springer-Verlag, 1975.



Kane, G. and Hawkins, D. and Leventhal, L., *68000 Assembly Language Programming*, Osborne/ McGraw Hill, Berkley, Cal., 1981.

Kernighan, B., and Plauger, P., *Software Tools in Pascal*, Addison-Wesley, 1981.

Newman, W. and Sproull, R., *Principles of Interactive Computer Graphics*, McGraw Hill, 1979 (Second Edition).

Tiberghien, J., *The Pascal Handbook*, Sybex Inc. , 1981.

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

# **L** The Macintosh Character Set

|   | 0            | 1            | 2  | 3 | 4 | 5 | 6 | 7   | 8 | 9 | A | B | C   | D | E | F |
|---|--------------|--------------|----|---|---|---|---|-----|---|---|---|---|-----|---|---|---|
| 0 | NUL          | DLE          | SP | 0 | @ | P | ` | p   | Ä | ê | † | ∞ | ¿   | — |   |   |
| 1 | SOH          | DC1          | !  | 1 | A | Q | a | q   | À | ë | ° | ± | ¡   | — |   |   |
| 2 | STX          | DC2          | "  | 2 | B | R | b | r   | Ç | í | ¢ | ≤ | ¬   | " |   |   |
| 3 | ETX<br>Enter | DC3          | #  | 3 | C | S | c | s   | É | ì | £ | ≥ | √   | " |   |   |
| 4 | EOT          | DC4          | \$ | 4 | D | T | d | t   | Ñ | î | § | Υ | f   | ' |   |   |
| 5 | ENQ          | NAK          | %  | 5 | E | U | e | u   | Ö | ï | • | μ | ≈   | , |   |   |
| 6 | ACK          | SYN          | &  | 6 | F | V | f | v   | Ü | ñ | ¶ | ∂ | Δ   | ÷ |   |   |
| 7 | BEL          | ETB          | '  | 7 | G | W | g | w   | á | ó | β | Σ | «   | ◇ |   |   |
| 8 | BS           | CAN          | (  | 8 | H | X | h | x   | à | ò | ® | Π | »   | ÿ |   |   |
| 9 | HT           | EM           | )  | 9 | I | Y | i | y   | â | ô | © | π | ... |   |   |   |
| A | LF           | SUB          | *  | : | J | Z | j | z   | ä | ö | ™ | ∫ | —   |   |   |   |
| B | VT           | ESC<br>Clear | +  | ; | K | [ | k | {   | ã | õ | ' | a | Á   |   |   |   |
| C | FF           | FS<br>↩      | ,  | < | L | \ | l |     | à | ú | " | o | Ã   |   |   |   |
| D | CR           | GS<br>▶      | —  | = | M | ] | m | }   | ç | ù | ≠ | Ω | Õ   |   |   |   |
| E | SO           | RS<br>▲      | .  | > | N | ^ | n | ~   | é | û | Æ | æ | Œ   |   |   |   |
| F | SI           | US<br>▼      | /  | ? | O | _ | o | DEL | è | ü | ø | ø | œ   |   |   |   |

Row and column headings are hexadecimal digits.

(Row16) + Column gives you the numeric code for the character.

The first 32 characters (DO-IF) and DEL (7F) are nonprinting control codes.

The shaded area is reserved for future use.



# Index



**A**

ABS, 92  
 active window, 17  
 actual parameter, 111, 113  
 addition, 37  
 address, 29  
 algorithm, 44  
 and, 48  
 animation, 206  
 area, 44, 45  
 argument, 86  
 array, 125-145  
 ARCTAN, 92  
 ASCII, 85, 87, 143, 144, 179  
 assignment operator, 31, 32, 37  
 assignment statement, 22, 31, 32, 36  
 atomic, 21

**B**

backspace, 11, 13  
 begin, 23, 29, 52  
 binary operators, 48  
 bit, 5  
 bit mapped, 5, 6  
 bomb, 81  
 boolean, 26, 47, 50  
 boolean expressions, 48-50  
 boolean operators, 47, 48  
 Boole, George, 47  
 break point, 83  
 bubble sort, 153  
 bug, 80  
 button, 8, 197  
 byte, 2

**C**

case, 154-157, 234, 235, 289  
 central processing unit (CPU), 2  
 char, 26, 85  
 character, 2, 27, 28  
 Check, 79, 272  
 CHR, 86, 87  
 circle, 98, 100  
 Clear, 271  
 Clipboard, 274  
 Close, 218, 290  
 COBOL, 89  
 colon, 29  
 column, 130  
 comment, 22, 23  
 compiler, 4, 5

compound interest, 61-63  
 compound statement, 52  
 Computational, 91  
 computer language, 4  
 Concat, 147  
 conditional expressions, 51  
 conditional test, 50  
 const, 41  
 constants, 40, 41  
 control variable, 56-58, 84  
 coordinate system, 69  
 Copy, (string function), 148  
 copy, 14, 271  
 COS, 92  
 Cosecant, 92  
 crash, 154  
 cursor, 10, 197  
 cut, 14, 271

**D**

data, 26  
 data types, 26, 29, 289  
 debugging, 77  
 decimal point, 27  
 Delete, (string function), 148  
 delete, 13  
 desktop, 6, 7, 10, 213  
 dialog box, 14, 15  
 difference, 177  
 disk drive, 4  
 display screen, 1  
 dispose, 238  
 div, 37, 38, 66, 67  
 division, 37  
 documentation, 22, 23, 275  
 do it, 84  
 Double, 90, 91  
 double click, 8, 14  
 downto, 61  
 draft quality print, 16  
 drag, 13  
 drawing window, 9-11, 69, 71, 185, 189, 274  
 DrawChar, 200  
 DrawString, 199  
 dynamic variable, 236-242

**E**

editing, 11, 13  
 ellipse, 98  
 else, 53-56  
 end, 23, 52

enumerated types, 94  
EOF (End of line), 221  
EOLN (End of line), 143, 222  
eject, 14, 15  
EraseOval, 98, 192  
EraseRect, 73, 74, 190  
EraseRoundRect, 192  
error messages, 317  
Everywhere, 272  
execution error, 81  
EXP, 93  
exponent, 27  
expression, 35-40  
Extended, 90, 91  
extended real, 90

## F

---

factorial, 159  
FALSE, 47  
field, 167-169  
fieldwidth, 33-35  
file, 213-231, 290  
file buffer, 215-219  
file menu, 8, 14, 16  
FillRect, 190  
Find, 272  
Finder, 213  
finger, 77  
floating point notation, 27, 35  
floppy disk, 4  
formal parameter, 111, 113  
for loop, 56-61, 63  
FORTRAN, 89  
FrameOval, 98, 192  
FrameRect, 74  
FramRoundRect, 192  
function, 86, 157-159, 160, 190

## G

---

Get, 217-220, 290  
GetMouse, 196  
GetTime, 175, 176  
global variable, 107-110, 113  
Go, 11, 78-80, 84, 272  
Go-Go, 83, 84  
graphics, 10, 185-210

## H

---

Halt, 79  
hardware, 1  
HideCursor, 197

high level language, 4, 5  
high quality print, 16

## I

---

icon, 6, 8, 213  
if-then, 49-56  
if-then-else, 53-56  
identifier, 14, 19, 21, 22, 29  
include, 149  
indent, 11  
infinite series, 63  
initialize, 30  
input, 1, 222  
input validation, 153, 250  
Insert, (string procedure), 149  
insert, 13  
insertion point, 10  
instant, 273  
instant window, 84  
integer, 26, 34, 89  
interest, 61-63  
interpreter, 4, 5, 77  
intersection, 178  
invert, 8, 13  
InvertCircle, 100  
InvertOval, 192  
InvertRect, 190  
InvertRoundRect, 192

## J

---

justify, 33

## K

---

K, 2  
keyboard, 1

## L

---

language translator, 4  
Length, 147  
line, 120-122, 186, 187  
LineTo, 121, 122, 165, 187  
LN, 93  
local variable, 107-109  
logarithmic functions, 93  
logical operators, 48  
logic error, 81  
LongInt, 89, 90  
low-level language, 4

**M**

machine language, 4, 5  
 MacWrite, 14  
 Maxint, 27  
 main memory, 2  
 memory, 3, 29  
 menu, 156  
 menu bar, 8  
 micro floppy, 4  
 mod, 37, 38, 66, 67  
 modem, 223  
 mortgage, 117-120  
 Motorola 68000, 2, 3  
 mouse, 1, 6, 195-197  
 Move, 186  
 MoveTo, 121, 122, 186  
 multiplication, 37  
 music, 284

**N**

nested for loop, 132  
 nested if statements, 55, 56  
 nested records, 173-175  
 New, 236, 237, 239  
 NewFileName, 224  
 Nil, 238  
 not, 48  
 Note, 284, 286  
 Notepad, 14

**O**

object code, 4, 5  
 observe window, 82-84  
 ODD, 92, 158  
 OldFileName, 223, 224  
 Omit, 148  
 Open, 216, 219, 220, 226, 290  
 operand, 39  
 operator, 37, 39  
 operating system, 5  
 or, 48  
 ORD, 86-88, 95, 96  
 ordinal type, 86  
 otherwise, 155, 157  
 output, 1, 23, 222  
 oval commands, 98-101, 191, 192

**P**

Page, 93  
 Page Setup, 271

PaintCircle, 98  
 PaintOval, 98, 192  
 PaintRect, 74, 190  
 PaintRoundRect, 192  
 parameters, 109-117  
 parentheses, 40, 49  
 paste, 14, 271  
 Pause, 79, 274  
 pen, 121, 192  
 PenMode, 193-195  
 PenPat, 192, 193  
 PenSize, 192  
 period, 23  
 perimeter, 44, 45  
 picture element, 69  
 pixel, 69  
 point, 69, 185, 186  
 pointer, 8, 10  
 pointer (^), 236-242  
 PointInRect, 202  
 Pos, 148  
 precedence, 39, 40, 48, 49  
 PRED, 88, 95  
 primary storage, 2  
 Print, 271  
 printer, 16, 222  
 printing a program, 16  
 procedure, 103-122, 159  
 program, 4, 19, 21, 22  
 programming language, 4  
 program pointer, 77  
 program window, 9  
 pseudocode, 44  
 Put, 217-220, 290

**Q**

QuickDraw, 71, 121, 185, 186, 308  
 Quit, 271  
 quotes, 27, 28

**R**

RAM, 3  
 random access, 214, 219, 220  
 read, 42-44, 143  
 readIn, 42-44, 143, 146  
 read only memory, 3  
 real, 26, 27, 34, 89, 90, 91  
 recalling a program, 15, 16  
 record, 165-175  
 record, array of, 170-173  
 record, variant, 233-236  
 Rect, 70-72, 187-189



rectangle, 44, 45, 187-189  
rectangle commands, 71-74, 190  
recursion, 159-165  
repeat, 151-153  
Replace, 272  
reserved word, 22, 292  
Reset, 79, 216, 220, 272, 290  
Revert, 271  
Rewrite, 216, 217, 220, 290  
ROM, 3, 185  
ROUND, 88, 89  
round cornered rectangle commands,  
    191, 192  
row, 130  
Run, 11, 79  
run menu, 77  
run time error, 81

---

**S**

---

Sane, 285, 286, 314  
Save, 270  
Save As, 270  
save a program, 14  
scalar type, 125  
scientific notation, 27  
scope of variables, 106, 111  
scroll, 15  
secondary storage, 4  
SectRect, 202  
Seek, 220, 290  
select, 8, 13, 14  
Select All, 271  
semicolon, 22, 29  
sentinal, 67, 128, 129  
sequential access, 214, 216, 218  
set, 176-180  
set operators, 177  
SetRect, 70, 73, 189  
SetDrawingRect, 189  
SetTextRect, 189  
shortcut, 14  
ShowText, 70, 189  
ShowCursor, 197  
ShowDrawing, 74, 189  
signed integer, 20  
SIN, 92  
software, 1, 4  
sorting, 153, 154  
sound, 284  
source code, 4  
source program, 4  
SQR, 91, 92  
SQRT, 92

standard quality print, 16  
statement, 22, 52  
Step, 79, 80, 83, 273  
Step-Step, 70, 80, 83, 84, 273  
StillDown, 197  
Stops-In/Stops-Out, 273  
stop rule, 160-163  
string, 26, 27, 28, 85, 125, 145-149,  
    289  
string comparison, 147  
string functions and procedures, 147-  
    149  
strong typing, 97  
structured programming, 105  
structured type, 125, 165  
subprogram, 103  
subranges, 96, 97, 126, 153  
subscript, 127, 130  
subtraction, 37  
SUCC, 88, 95  
syntax, 19  
syntax diagram, 20  
syntax error, 77, 79, 80  
SysBeep, 93

---

**T**

---

tab key, 17  
tag field, 234, 235  
tangent, 92  
TextFace, 200-202  
text file, 214, 221-224  
TextFont, 200  
TextSize, 200  
text window, 9, 10, 24, 68-70, 222,  
    274  
TickCount, 93  
TicTacToe, 132-142  
toolbox, 3, 69, 70, 93, 195  
top down programming, 105  
trash can, 8  
trigonometric functions, 92  
triple clicking, 14  
TRUE, 47  
TRUNC, 88, 89  
two dimensional array, 130-142  
type, 94  
type checking, 236  
Type Size, 274

---

**U**

---

UCSD Pascal, 90, 289  
unary operators, 48



undeclared identifier, 81  
underscore, 21  
union, 177  
unit, 185  
unsigned integer, 20  
user-defined data types, 94-97

**V**

value parameter, 113-117  
var, 29, 111  
variables, 29, 31, 41  
variable parameter, 111, 112, 115-117  
variant record, 233-236  
video display, 5, 6

**W**

WaitMouseUp, 197  
while loop, 64-67  
window, 9, 10  
with, 168-170  
world symbol, 22  
write, 23-25, 33-35  
writeln, 23, 25, 28, 29, 33-35

**X**

XpwrY, 285

## Related Resources Shelf

### **MacPaint: Drawing Drafting Design**

Susan Schmieman

Create your own masterpieces with MacPaint, Apple Computer's graphics program. Both novices and experienced Mac users can discover the magic of drawing with the help of this easy-to-read, well-organized guide fully packed with clear, imaginative illustrations.

☐ 1985/197pp/paper/D648X-5/\$16.95

### **Macintosh: The Definitive User's Guide**

John M. Allswang

With this easy-to-follow handbook, you'll learn all there is to know about your machine and what you can do with it. Discover the Macintosh's best-kept secrets by finding out how to tap all its capabilities and characteristics.

☐ 1985/244pp/paper/D6498-2/\$16.95

### **Microsoft BASIC for the Macintosh**

Larry Joel Goldstein, David I Schneider

A two-books-in-one approach to learning on the Macintosh! The first part serves as a tutorial, introducing beginners to the fundamentals of Microsoft BASIC for the Macintosh. The second part offers a detailed reference manual that describes all the commands of Microsoft BASIC.

☐ 1985/561pp/paper/D6625-0/\$19.95

### **Business and Home Applications for the Macintosh Using Microsoft BASIC**

Stan Schatt

This book for beginning Macintosh users offers business, education, and home use programs, using Microsoft BASIC. It's a practical guide that offers fast, easy, and useful home and business applications.

☐ 1985/201pp/paper/D4037-0/\$14.95

To order, simply clip or photo copy this entire page, check off your selection, and complete the coupon below. Enclose a check or money order for the stated amount. *(Please add \$2.00 postage/handling per book plus local sales tax.)* Or call toll-free 800-638-0220; in Maryland, 301 262-6300.

Mail to:

**Brady Communications Co., Inc. • Dept. TS • Bowie, MD 20715**

Name

Address

City/State/Zip

Charge my credit card instead: ☐ MasterCard ☐ VISA

Account #  Expiration Date

Signature

Dept. Y

Y0510-BB(5)

*Prices subject to change without notice.*



Prepublication reviewers say:

*"...an invaluable addition to a Macintosh owner's library!"*

*"...strongly organized with exceptionally clear and fun programs which do much to speed up the MacPascal learning process!"*

## **THE MACPASCAL BOOK**

**Paul Goodman and Alan Zeldin**

Now—a text that teaches Pascal programming for the Macintosh! This comprehensive tutorial covers the entire breadth of the Pascal language and the MacPascal system, with detailed descriptions of file usage and I/O. Includes:

- An introduction to computer hardware and software concepts as they appear on the Macintosh
- Step-by-step instructions on how to enter, edit, and run a simple program in MacPascal
- A comprehensive look at the Macintosh graphics coordinate system, including animation programs
- Plus complete chapters on debugging, data types, functions, procedures, and much more!

## **CONTENTS**

Computer Concepts • Using MacPascal • Pascal Fundamentals • Pascal Structures • Debugging a MacPascal Program • More on Data Types • Procedures • Arrays and Strings • More on Structures • A Formal Look at Graphics • Files • Variant Records and Pointers • A Look at an Application—The Checking and Savings Account • Appendix A—Selected Exercise Answers • Appendix B—Menu Summary • Appendix C—Documenting a Program • Appendix D—Sound and Music • Appendix E—Differences Between MacPascal and UCSD Pascal • Appendix F—MacPascal Reserved Words • Appendix G—MacPascal Syntax Diagrams • Appendix H—List of Quickdraw Routines • Appendix I—List of Sane Functions and Procedures • Appendix J—MacPascal Error Messages • Appendix K—Bibliography



ISBN 0-89303-644-7