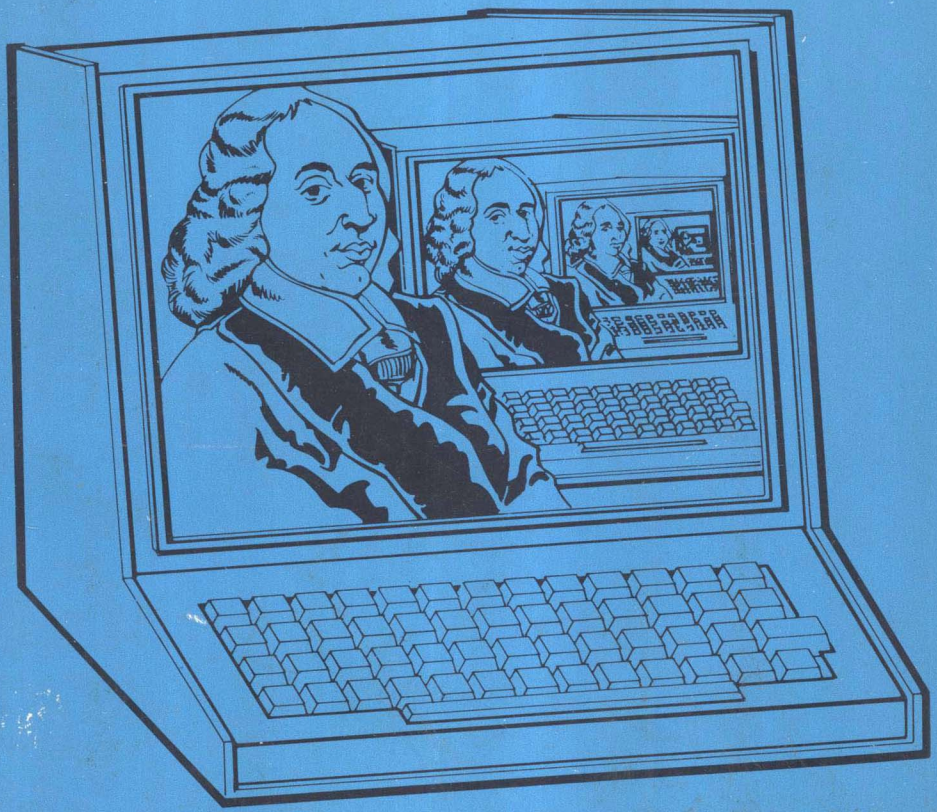
Pascal Programming Structures for Motorola Microprocessors



George W. Cherry

FOROLA
s in Solid-State Electronics

Pascal Programming Structures
for Motorola Microprocessors

George W. Cherry

Reston Publishing Company, Inc.
A Prentice-Hall Company
Reston, Virginia

MOTOROLA
Series in Solid-State Electronics

Library of Congress Cataloging in Publication Data

Cherry, George W.
Pascal programming structures for Motorola microprocessors.

(Motorola series in solid-state electronics)
Includes index.
1. Pascal (Computer program language) 2. Microprocessors--Programming. 3. Structured programming.
I. Title. II. Title: Motorola microprocessors.
III. Series.
QA76.73.P2C4G7 001.64'24 81-15327
ISBN 0-8359-5465-X

10 9 8 7 6 5 4 3 2 1

Printed in the United States of America

Preface

Beginning students in programming have several kinds of difficulties in writing well constructed and correct programs:

1. Getting started
2. Finishing
3. Writing statements free of syntactical errors

A helpful textbook on programming assists the student in these three areas. This book was written in accordance with this idea. The book helps the student to get started by illustrating the top-down strategy of design. It helps the student to complete his design by illustrating step-wise refinement. And it helps the student to avoid syntactical errors by careful descriptions and illustrations of Pascal syntax. I believe the result will be that students using this book will write error-free and pleasing programs earlier in their courses.

The book emphasizes good reading and writing skills. One of my hopes is that the book's emphasis on clear style will infect the reader with an enthusiam for writing and reading well written computer programs. A bibliography tells the student where he can find many more well written Pascal programs.

I have chosen Pascal for this textbook because of its outstanding balance of data structuring facilities, control structures, and size. Pascal facilitates the objectives of an introductory programming course better than any other language I know. Pascal is a practical as well as educationally significant language. Efficient compilers for Pascal are available for virtually every contemporary computer. The present trickle of ads for programmers who know Pascal should grow steadily during the 1980s. Students who learn to program in Pascal according to the principles expounded in this book should write better BASIC, FORTRAN, and COBOL programs when their jobs require these older languages.

The first two chapters of the book are different in style and intent from the rest of the book. The purpose of these chapters is to introduce the rank beginner to simple program statements and simple, complete computer programs. It's beneficial for students to see and comprehend a handful of whole programs before getting into details. This is by no means a new approach; but it's sound and sadly underused.

The student who already knows another programming language and wants to "pick up" Pascal could skip Chapter 1 completely and read Chapter 2 rapidly. This student may find the rest of book somewhat verbose; but it's my experience that students don't save time when an author leaves them half guessing at some syntactical point or semantic possibility.

The book starts at a level suitable for the rank novice; but it covers ground fairly rapidly as the student's knowledge grows. The book describes the complete Pascal language. It concludes with records (Chapter 8), dynamically created data structures (Chapter 9), files (Chapter 10), and sets (Chapter 11). (However, files in the context of the standard input and output operations are also covered in Chapter 4.) We leave sets until last because, while elegant, they are more peculiar to Pascal, and not essential.

I see the book's application in three settings:

1. A textbook for a one-semester course entitled something like "Introduction to Computer Programming." Such a course would probably assign the first seven chapters and thus cover all the simple data types, all the control structures, subprograms, and arrays. The order of the chapters on arrays (Chapter 6) and subprograms (Chapter 7) can be switched (by deferring the array examples in the chapter on subprograms). At any rate, we introduce the concept and some simple illustrations of subprograms long before the formal chapter on subprograms.

This would be a very respectable introductory course. It would give the students possession of a language approximately equal in power to BASIC and FORTRAN. The instructor might, of course, elect to omit the sections on recursive subprograms, procedures and functions as parameters, and any other section he deemed too advanced.

2. A second course in programming entitled "Data Structures" or "Information Structures." In this course the instructor would assign Chapters 6-11 and the students could use the earlier chapters for reference. Pascal is an exquisite language for teaching and learning data structures. For this course the instructor would no doubt want to augment this book with his own notes or one of the excellent books covering data structures (Professor Wirth's own "Algorithms + Data Structures = Programs" or Horowitz and Sahni's "Fundamentals of Data Structures," for example).

3. A self-study text for the computer professional or hobbyist who wants to learn Pascal for professional or cultural reasons.

I have taken very seriously the careful explication of Pascal's syntax. It's gratuitous frustration for a student to wrestle with a malfunctioning program because his textbook failed to elucidate some syntactical banana peel it's easy to slip on. I know of one student who spent three hours trying to debug a program because his textbook did not explain Pascal's peculiar behavior while reading numbers and looking for eof = true. Where there are ambiguities or disputes about a particular definition, I have appealed to the proposed BSI/ISO/ANSI* standard as well as Pascal's de facto standard, Jensen and Wirth, 1974.

All the major and many of the minor programming examples have been compiled and run on a computer.

The book requires very little mathematical knowledge of the student beyond elementary algebra.

I have taken illustrative problems from the fields of business, computer science, elementary chemistry, information theory, psychiatry, psychology, typesetting, word processing, and others.

Finally, for those who are interested in such things, I should like to say a little about the production of this book. The author prepared the text on his personal microcomputer and "set the type" himself with the same microcomputer, using a popular serial character printer. (The only type the author did not personally set was the Bauhaus Medium display type used for the title page and chapter titles.) This kind of composition system dissolves some of the technologically obsolete boundaries between author and copy editor, author and typesetter, author and proofreader. Therefore, I must bear fuller responsibility than usual, for better or for worse, for errors occurring in this book.

I am particularly pleased to acknowledge the expertness of Ellen Cherry, my wife and very fortunately my production editor at Reston Publishing Company. Ellen skillfully guided me through the many shoals of bookmaking.

* BSI: British Standards Institution
  ISO: International Standards Organization
  ANSI: American National Standards Institute

George W. Cherry
Reston, Virginia

PREFACE TO THE THIRD PRINTING

I am grateful to my publisher and readers for the opportunity to include in this printing several improvements and corrections. I especially want to express my gratitude to those thoughtful readers who found and reported errors or obscurities in the earlier printings.

January 1981 George W. Cherry

Contents

1. Introduction to Programming

A computer program is a sequence of instructions to a computer processor to perform useful actions on significant objects. The significant objects are often--but by no means always--numbers. For example, there are extremely useful computer programs for helping authors, secretaries, editors, and typesetters to create, prepare, edit, and typeset English text. In these cases the objects are letters, numbers, punctuation marks, words, sentences, and paragraphs; and the useful actions are insert, delete, move, justify (meaning to make the margins line up, as in the case of both margins of this book), paginate, and print. This book, for example, was composed, edited, and printed with the aid of such a computer program. There are several excellent word-processing and page-formatting programs written in the Pascal programming language.

The significant objects can also be lines, curves, characters, and other elements of figures; examples of the useful actions are position, scale-up (or scale-down), rotate, extend, change color, move, erase, connect, and print. Objects and actions like these are used in computer programs for computer-aided drafting, drawing, and design. An example of the output of such a program is Figure 1.1. A relatively short and simple Pascal program generated this intricate figure, called a Sierpinski curve.

Our final example of significant objects is records that are arranged in lists. The useful actions in this case are manipulations on the data in the records; this is called "list processing." An important example would be an insurance company's list of records of its policy holders. Useful actions would be to print the names of all policy holders who had more than three claims in the last five years; to print reminder notices to all policy holders whose premiums are 30 days overdue; to send information about life insurance to all policy holders who have fire insurance with the company and no life insurance; or to send information about automobile insurance to all the policy holders who have life insurance and no automobile insurance. Unlike the programming languages Fortran and BASIC, which are somewhat deficient in features that facilitate list processing, Pascal has language features allowing the full generality of list processing.

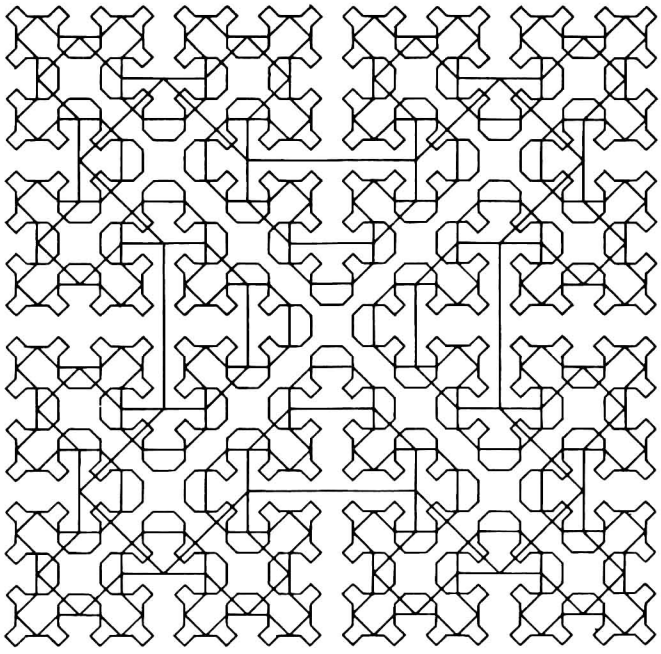


Figure 1.1 Sierpinski Curve

We call the significant objects on which the computer performs useful actions, data. We have seen that the family of data contains more types of objects than just numbers. (It also includes more than one type of number, as you will learn.) The first ingredient of a well written program is a thorough and careful description of the types of data on which your program acts. There are two kinds of data in a computer program: data whose values cannot change during the execution of the program, called, for obvious reasons, constants (such as the number of ounces in a quart, 32); and data whose values can change, called variables. Variables are, of course, where most of the program's action is. The value of a variable can change; but in the Pascal language a variable's data type cannot change. If a programmer defines a variable to hold a string of characters, he must not later treat that variable as though it held a number representing a component of the national debt. Therefore, every variable in a Pascal program has a distinct and unchanging data type associated with it; and this data type determines both the values the variable can assume and the operations the program may perform upon it. An immediate advantage of this data type stability is that we may give every variable a meaningful name like Employee, HoursWorked, HourlyRate, CitationsInStock, OptionalEquipment, SalesToDate, and so on; and these meaningful names are more likely to stay meaningful during the existence of the variable.

A computer programming language is a formal means of describing the significant objects (the data) and the useful actions that the computer processor should perform on the data. Since computers don't "think" the way we do, there is always some compromise when we human beings try to communicate with a computer. We prefer to speak in an abstract and problem-oriented natural language. We would like to say something to the computer like "Keep a record of the hours and hourly rate, overtime, and bonuses of each employee and print out their paychecks each month," or "Keep an inventory of all the new automobiles we have in stock and tell me right now how many air-conditioned, manual transmission, two-door Citations we have in stock," or "Keep a record of the sales of all salesman and print out their names and sales-to-date in descending order."

Unless you believe the movies in which space travelers carry on colloqial dialogues with talking computers, you probably already know there's a vast gulf between what we would like to say to the computer and what the computer can understand. The computer's objects, after all, are only strings of binary digits, or "bits," like 11010001, 00100101, 11100111, and so on, not the consequential objects (like Employee, HoursWorked, and HourlyRate) with which we associate names, meanings, and data types. And the computer's native actions are primitive and elementary, not related in any way we can readily see to the problem-solving actions we've been discussing. Furthermore, the computer hardware requires all its nondescript data and its primitive instructions to be expressed as strings of bits. If we didn't already know that computers are useful, we might despair that they ever could be. How do we bridge this vast gulf between the human problem and the computer's reliable and fast but simplistic mechanisms? One aid in bridging the gap is an appropriate programming language. Ideally, a programming language should make it easy for us to express our problem solutions; but the language should also be easy to translate into those machine language strings of ones and zeros.

Fortunately, since the installation of the first useful digital computers in the 1950s, the compromise in communication has been continually shifting in our--the human being's--favor. The reason is the successful effort of computer scientists to develop high-level, problem-oriented languages and efficient computer programs to translate these languages into the strings of ones and zeros required by computer processors. Consequently, we don't have to worry about the ones and the zeros, the machine's internal language, and other esoteric matters like that. Someone must, of course, and you may want to learn about that kind of programming also--it's called machine-level or machine language programming. But this is a book about programming in a high-level, problem-oriented language. The computing community often uses one of the following abbreviations for this kind of programming: HLL (High-Level Language) or POL (Problem-Oriented Language) programming.

A computer program called the compiler (and sometimes the interpreter) makes the translation of your HLL program into the computer's internal language. Thus, the computer helps solve the

communication problem which it created. You use the computer to translate your language into its language. Therefore, when you approach the computer for the first time with your first Pascal program, you will be meeting the compiler and asking it to compile (translate) your program. A good compiler will do more than translate your program for you. It will also rigorously check your program's adherence to Pascal's data typing and syntax rules and make whatever checks it can of your program's logical consistency. A good compiler is the HLL programmer's best friend. Compilers make possible programming enterprises that would probably be quite impossible without them. We will make frequent reference to the compiler throughout this book. Here's a definition of compiler from the 1977 edition of Webster's New Collegiate Dictionary: "a computer program that translates instructions in a higher-level symbolic language (as COBOL [Fortran, BASIC, PL/I, or Pascal]) into machine language."

A COMPUTER PROGRAM IS LIKE A RECIPE...

The recipe for a cake, the instructions for knitting a sweater, or the directions for building a birdhouse or assembling an electronic kit have a logical format. Consider the following recipe:

BAKED ZITI (Serves 4)

Ingredients:
1 (8-ounce) package ziti
2 cups Italian tomato sauce
1 cup shredded mozzarella cheese

Actions:
1. Cook ziti according to spaghetti recipe on page 343.
2. Prepare Italian tomato sauce according to recipe on page 186.
3. Combine ziti, tomato sauce, and 1/2 cup shredded cheese in 2-quart casserole.
4. Sprinkle remainder of cheese on top of casserole.
5. Heat casserole uncovered on full power for ten minutes or until cheese is melted and sauce is bubbly.

This recipe is a program for preparing a pasta casserole; it has many

parallels with a computer program. Like the recipe, a Pascal program must start with a heading. Like the recipe a Pascal program must follow the heading with a list of its "ingredients," the objects upon which the program operates.

Notice that for the sake of compactness and readability the recipe refers to subrecipes (or procedures) on other pages of the cook book: the procedures for cooking the ziti and preparing the Italian tomato sauce. Indeed, the procedure (for preparing the tomato sauce) on page 186 of the cookbook is actually longer than this recipe for Baked Ziti. The single instruction in action 2 in the main recipe stands for the many declarations and actions defined in the Italian tomato sauce procedure on page 186. Many well organized cook books declare a set of useful procedures, functions, and subrecipes which the author invokes again and again. For example, procedures for drawing a bird and stuffing and trussing a bird may appear at the beginning of the poultry section; later in the cookbook, recipes for wild duck, turkey, and partridge invoke these predefined procedures. The author will give gravy and sauce subrecipes and then invoke them in main recipes by short-hand phrases such as "White Sauce III, page 285." Pascal offers two forms of subprograms (procedures and functions) which are analogous to these subrecipes. They serve the same purpose: they make the main program more compact, readable, and comprehensible.

A particularly useful kind of instruction in a recipe is one that makes the actions of the cook contingent on the condition of the item under preparation. Statement 5 above contains such an instruction: the recipe instructs the cook to heat the casserole "until cheese is melted and sauce is bubbly." There are many such contingent actions in cooking recipes; they ensure repetition of an action (heating, in our example) until a condition is achieved (melted cheese and bubbly sauce). Other examples from a popular cookbook are: "simmer celery until tender"; "beat the batter until it is smooth"; "whip the egg whites until they stand in peaks." Pascal has two kinds of instructions for controlling contingent program repetitions (see Chapter 5 in Contents). Borrowing the Pascal keywords they are:

```
while "the batter is lumpy" do
   "beat the batter."
```

and

```
repeat
   "simmer the celery"
until "the celery is tender."
```

The above is pidgin Pascal. Here's some real Pascal:

```
repeat
    X := X / 2
until X < 1
```

What does it do? First of all, every trade and profession has special jargon and symbols. Computing (and cookery!) is no exception. There's one of these special symbols in the above statement: ":="; it's called the assignment operator. The assignment operator enforces a two-stage process:

1. Evaluate the expression on the right-hand side; i.e., find the value of X divided by 2.

2. Assign this value to the variable whose name appears on the left-hand side.

How do you say "X := X / 2"? Programmers usually simply say "X equals X over two." Of course, this is not an accurate translation of the assignment operator; and it's sheer arithmetical nonsense unless X is zero. But you're invited to say it the easy way. Pascal will always try to remind you that the assignment operator is not the same as "=" or "equals"; that's precisely why Pascal uses ":=" for the assignment operator. If you wanted to be strictly correct (and sound rather pedantic and stiff!), you might say "replace the old value of X with the old value of X divided by two."

Let's go back to our original question: what does this **repeat-until** statement do? It repeatedly halves the value contained in the variable X until that value is less than one. If the initial value in X is 10, then the sequence of values generated by the **repeat...until** loop is: 10.0, 5.0, 2.5, 1.25, 0.625; and the final value assigned to X by this process is 0.625.

A simpler kind of repetition in a recipe is noncontingent: the action is repeated for a fixed number of times or for a given duration. Examples are: "stir mixture ten times"; "blend for six minutes"; "rinse in clear, cold water three times." Pascal has a control structure analogous to this. Here's some pidgin Pascal translating two of the above instructions.

```
for J := 1 to 10 do
   "stir the mixture";

for J := 1 to 3 do
   "rinse in cold, clear water";
```

And here's some real Pascal.

```
for J := 1 to 4 do
   writeln('This is a test.');
```

This Pascal instruction prints (writes a line) "This is a test." four times on the output device (printer or video console).

The nature of a cooking or programming problem may be such that, depending on the condition of the object, the cook or the processor should or should not take some action. From a recipe book we have: "pare

apples only if the skins are very tough" and "if the caramel is sugary, add more cream and boil again." In pidgin Pascal these instructions would be:

```
if "skins are very tough" then
   "pare the apples";

if "caramel is sugary" then
   begin
      "add more cream";
      "boil again"
   end;
```

Note in the second example that we enclose the compound statement "add more cream; boil again" with **begin** and **end.** A compound statement is two or more consecutive statements. With only one exception, we use **begin** and **end** to bracket compound statements in Pascal. You will see that this bracketing, which allows us to aggregate many simple statements into a delimited unit, enhances the convenience and clarity of writing and reading Pascal programs.

Sometimes the nature of the problem is such that, depending on the condition of the object or the option selected, the processor must choose one of two different actions. From a candy recipe that offers the two-way choice of making old-fashioned fudge or penuche, we have the following structured control statement.

```
if "candy = old-fashioned fudge" then
   begin
      "combine 2 cups granulated sugar, 5 tablespoons cocoa,
      and one-quarter teaspoon salt in mixing bowl";
      "stir in thoroughly 1 tablespoon light corn syrup and
      1 cup milk"
   end
else {"candy = penuche, so"}
   begin
      "combine 1 cup granulated sugar, 1 cup brown sugar,
      and one-quarter teaspoon salt in mixing bowl";
      "stir in thoroughly 2 tablespoons light corn syrup and
      1 cup milk"
   end;
"Add 3 tablespoons butter";
"Cover";
for minutes := 1 to 5 do
   "microwave at high setting";
```

This control structure directs the processor to execute either the first compound statement or the second compound statement. In either case (fudge or penuche), the program directs the processor to execute the last three statements because these actions are common to the making of both fudge and penuche.

Sometimes a problem requires us to chain **if** statements together. From a recipe book we have: "if mixture is too thin, add more milk powder; if mixture is too thick, add more water." Using Pascal reserved words and syntax we have.

```
if "mixture is too thin" then
   "add more milk powder"
else if "mixture is too thick" then
   "add more water"
```

The chaining of if statements by the **else if** phrase is so common in programming, that the new Department of Defense programming language, Ada, proposes a reserved word **elsif.**

Some recipe books give menus for every day of the week or every day of the month. The problem here is to select, according to the value of the variable today, one of many actions. Pascal provides the **case** statement to deal concisely with this type of selection.

```
case today of
   Monday :
      "Prepare menu 1";
   Tuesday :
      "Prepare menu 2";
   Wednesday :
      "Prepare menu 3";
   Thursday :
      "Prepare menu 4";
   Friday :
      "Prepare menu 5";
   Saturday :
      "Prepare menu 6";
   Sunday :
      "Prepare menu 7"
end.
```

This control structure will cause the processor to: prepare menu 1 when today is Monday; prepare menu 2 when today is Tuesday; prepare menu 3 when today is Wednesday; and so on.

NEED FOR CARE AND PRECISION IN WRITING COMPUTER PROGRAMS

Although there's an analogy between a cooking recipe and a computer program, there are also distinct differences. Programs for computers tend to be much longer and more intricate than cooking recipes; and computer programs are processed by ignorant and rigid automata, whereas cooking recipes are usually processed by relatively perceptive, versatile, and knowledgeable human beings. Therefore, the author of a

computer program must exercise a great deal more care and precision in the preparation of his instructions.

The cook, with his knowledge of cookery, the context, and the probable intention of the recipe's author, can compensate for some vague directions, ambiguities, and even downright errors. But the compiler, which translates the high-level instructions into the computer processor's language, and the processor itself, have a very restricted "understanding"; they cannot compensate for the short-comings or ambiguities of the careless programmer. Since the computer system cannot really understand your intention (even though it might be patently clear to any other human being), the system will translate and execute your data descriptions and instructions in a robotic, literal way. The compiler which translates your statements is also extremely fussy about spellings and syntax. Like any functionary low in the cognitive hierarchy, the compiler is very rule bound: it may reject a tightly reasoned and beautifully structured program because of one missing semicolon. You must quickly learn that the computing system is a dutiful automatic servant but a terrible professional peer.

You also write programs for your professional peers who may have to understand your program in order to modify it or correct an error that crops up in it long after you've finished it. Since computer programs can be very large and complex, you should take special pains to make your program clear and readable to other programmers. Also, you may want to understand your own programs six months or six years from the time you write them. Many programmers cannot understand their own programs a few months after they write them.

You write your programs for two kinds of "agents": human beings like yourself who possess rationality and intuition but whose rationality is limited and whose intuition is fallible; and the computer itself which possesses a very high degree of responsiveness but a responsiveness that is robotic, literal, and finicky.

These agents require that you write your programs with professional care and precision.

NAMES (IDENTIFIERS) IN PASCAL

As an example of the programmer's writing for two audiences, we will take the formation of names or identifiers in Pascal. The programmer must form an identifier for every constant synonym, nonstandard data type, variable, procedure, and function in his program. He must also form a name for his program. How should he do this? First, he must comply with Pascal's syntax rules for identifiers. Second, he should obey the programming injunction to choose mnemonic names. A mnemonic name conveys meaning; it reminds you of the thing it stands for. The reader of your program is on a quest for meaning. Assist him!

Here's an example of a program that uses syntactically legal but semantically uncertain identifiers. What does it mean? What does it do?