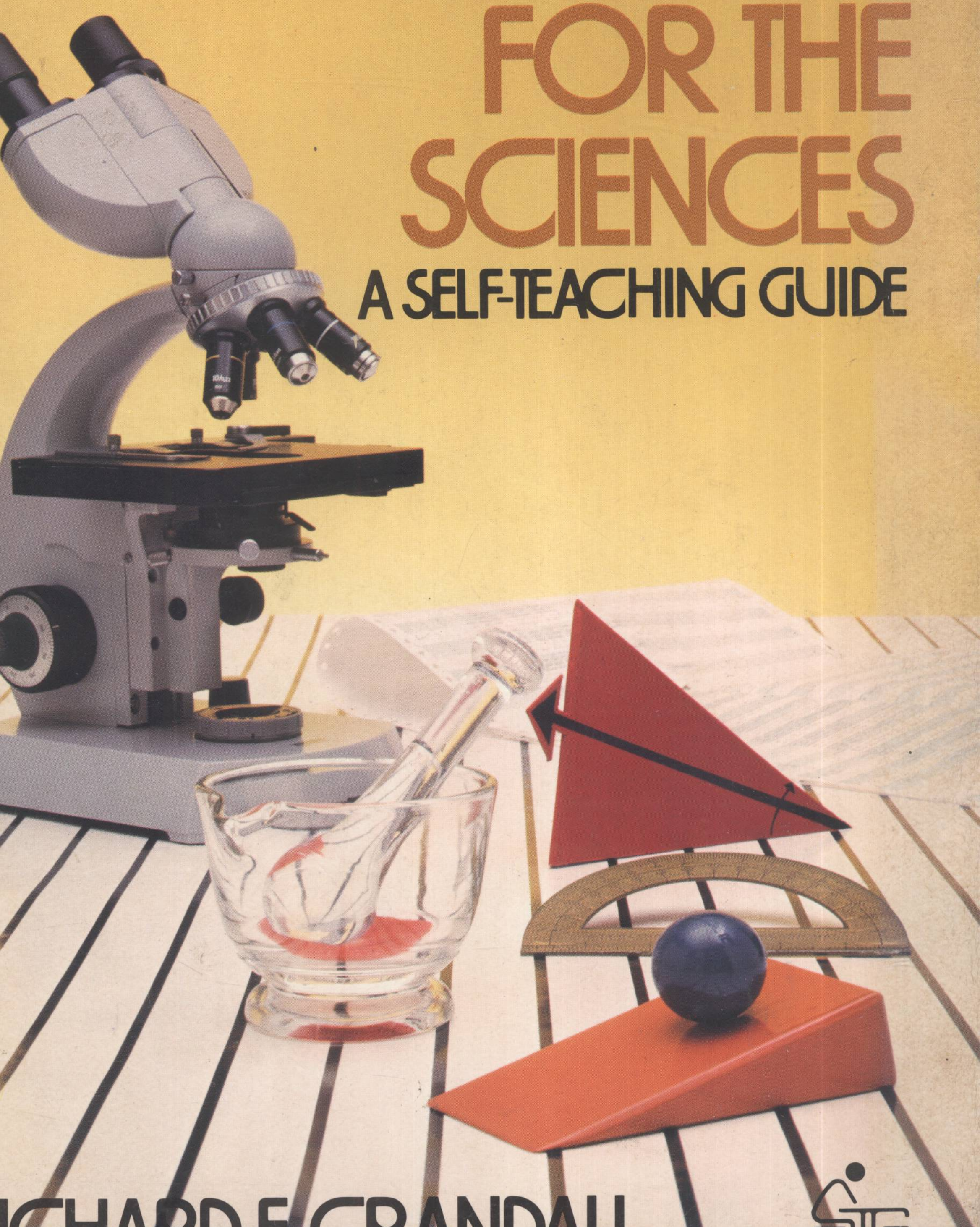
PASCAL APPLICATIONS FOR THE SCIENCES

A SELF-TEACHING GUIDE



ICHARD E. CRANDALL

STG

Pascal Applications for the Sciences

Richard E. Crandall

Department of Physics
Reed College
Portland, Oregon

A Wiley Press Book

JOHN WILEY & SONS, INC.
New York · Chichester · Brisbane · Toronto · Singapore

Pascal Applications for the Sciences

Preface

Pascal Applications for the Sciences has evolved as a result of a common dilemma facing teachers of science: How can a diverse group of students be taught scientific programming in a minimum amount of time? In 1978, the Reed College Physics Department encountered such a dilemma when they decided to adopt Pascal as the language of choice for physics courses. Even at the introductory level, students have diverse approaches to science and may be committed to a specific field such as biology, physics, medicine, or chemistry. The teachers had to resolve two questions: How can these students be brought to a level of scientific programming that is adequate for basic applications in their chosen disciplines? How can this be done without taking too much time away from the normal course of physics study? The dilemma was compounded by the fact that, even though some marvelous student assistants lent their talents to the Pascal sessions, the average introductory student had to spend considerable time alone at a terminal, in self-study mode.

This book incorporates what I have learned over a 5-year period about how students of scientific programming prefer to pace themselves. The course materials for the Pascal sessions at Reed College have been refined, tested, and amplified to provide a means by which students can acquire scientific programming skill in a short time. The book is for scientists and students of science who wish to achieve the expertise needed to solve problems in their particular disciplines. This book is not a traditional computer science book—it is a guide for any scientist. It contains a review of Pascal structure, but it focuses on applications. For this reason, I recommend that students use a standard Pascal text for details on Pascal syntax and structure. In the original college sessions, the materials in this book were used together with a standard Pascal text. This combination proved so effective that serious students were able to achieve good levels of expertise in one school quarter.

Pascal Applications for the Sciences has the following sequence: Pascal review, mathematical programming, equation solving, statistics, and graphics. The book then branches into four advanced chapters, with applications for mathematics, chemistry, physics, and biology. All the chapters include exercises that cover typical applications, as well as exercises of an exploratory nature.

The five appendixes contain scientific libraries as Pascal include files. These libraries have computer packages students will find useful in their studies; they cover graphics, matrices, statistics, special functions, and dynamical models.

Throughout the book the sequence of text—exercises—text—exercises—and so on is followed. I have found that there is a limit to how long a programmer can read text material before an exercise series is appropriate. The length of each text segment has been designed accordingly. For those facets of study I take to be more important, there are noticeably more exercises following.

Above all, *Pascal Applications for the Sciences* is designed to provide you with efficient, unified training. Scientists of today have so much to learn in their chosen areas that the acquisition of programming skill should not usurp an inordinate amount of time away from traditional scientific pursuit. You will have been successful with this book if you can achieve a certain independence from it. It is my hope that you will eventually be able to address any problem in your field, using the book only in matters of reference.

I would like to thank the introductory students whose feedback over the years has been indispensable for this project. I also wish to thank the instructors R. Henley, R. Whitnell, S. Swanson, W. Wood, and T. Abbott whose excellent works are included in this book. I am indebted to faculty members D. Hoffman, R. Reynolds, J. Delord, R. Bettega, N. Wheeler, D. Griffiths, J. Buhler, R. Mayer, T. Dunne, and J. Dudman for programs, corrections, and general support of the project. I thank computer colleagues C. Green, G. Schlickeiser, E. Roberts, G. Ross, M. Penk, and D. Basin for their gracious offerings of expertise. I especially thank S. Stearns for allowing me to transfer his insightful course materials for Pascal as it pertains to biology. I am grateful to Intel Corporation, Hewlett-Packard, Tektronix, and Apple Computer personnel for support in the form of equipment, encouragement, and general interest in the project. I am indebted to A. Marcus, M. Blair, R. Kilgore, R. Raber, M. Lindquist, and C. Delord for their inestimable aid in generation of the manuscript.

Richard E. Crandall
Portland, Oregon

How to Use This Book

Pascal Applications for the Sciences is designed to lead you from a novice level of scientific programming to a level of expertise at which you can routinely solve problems in your field of science. If you need a review of Pascal, begin with Chapter 1. If you already have some Pascal skills, look over the exercises in Chapter 1 to make sure that you can do them, then move on to Chapter 2 where applications begin. If you are already an experienced scientific programmer, try the more difficult exercises in Chapters 2 to 5 and the material in the advanced Chapters 6 to 9.

Each chapter has the following structure:

Text
Exercises
Text
Exercises
. . . .

Throughout the book the theme is "explore!" and in this spirit many of the exercises, notably the final ones of each exercise section, have been included as challenges for your creative abilities. There are references at the end of the book for most chapters so that you may pursue the scientific concepts discussed in the text.

You will notice that instructions in Chapters 6 to 9 are less direct and that the burden of creativity is passed to you. Many of the exercises in these chapters have the flavor of projects as opposed to test questions.

Good luck with Pascal and with science!

Contents

1 Pascal Review

These machines have no common sense; they have not yet learned to "think," and they do exactly as they are told, no more and no less. This fact is the hardest to grasp when one first tries to use a computer.

Donald E. Knuth
The Art of Computer Programming

GETTING STARTED

Pascal can be used in a great variety of ways such as to display data, to analyze data, to verify theoretical predictions, and to suggest new lines of scientific thought. It is suitable for applied as well as theoretical science. Many books only cover the abstract features of Pascal, such as detailed syntax rules and algorithm structure. This book, however, emphasizes the *utility* of the language. In order to benefit from the examples and exercises, it is important that you first obtain a working knowledge of Pascal. Since this book teaches you to instruct your computer to perform a wide variety of tasks, you must learn how to tell your computer exactly what you want it to do.

Exercise

Learn how to edit programs on your computer system. Compile these programs and arrange your files. Your own system's documents are best for this. Use a standard Pascal text, such as the ones listed in Chapter 1 references, for matters of detailed syntax. It is a good idea to read this chapter with a standard text at your side. If you can successfully do the exercises in this chapter, you are ready to move to Chapter 2 where essential mathematical applications begin.

ELEMENTARY SYNTAX

Most Pascal references contain *syntax diagrams*, which are roadmaps of the overall program and show graphically what structures are legal. Some programmers use references that have all relevant Pascal syntax diagrams; some never use them. In any case, they are useful in the preliminary stages of understanding. This chapter presents some diagrams that frequently occur in application programs.

The most important diagram is for the *program* itself (Figure 1.1). The syntax diagram in Figure 1.1 shows how to start editing a Pascal program. You follow the diagram from left to right and always start with the word "program."

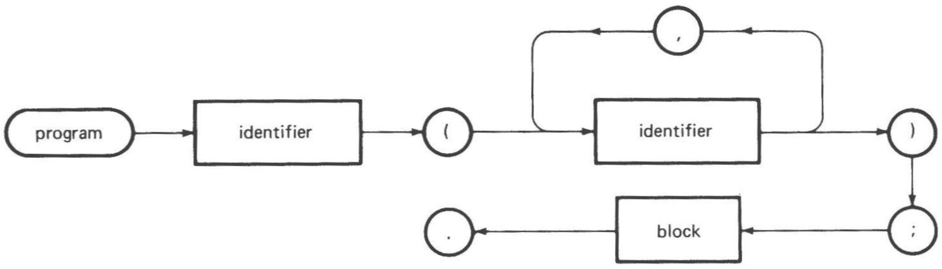


Figure 1.1

The word "program," enclosed in an oval box, is a *keyword,* or *reserved word,* in Pascal. The circled items are *operators,* which are Pascal's punctuation symbols. Items in square boxes are defined by other syntax diagrams. As you will see, what is normally thought of as a programming task will involve almost exclusively a structure called the *block.*

Identifiers are Pascal's words for labeling various objects. For example, the first identifier on the left in Figure 1.1 will be the name of the program. The identifiers following it in parentheses are called *file identifiers.* Generally, these take the form of some combination of the two words "input" and "output".

An identifier has its own roadmap. The diagram in Figure 1.2, for example, means that an identifier is any sequence of letters and digits beginning with a letter. Thus "x", "x5", "voltage", and "position" are legitimate Pascal identifiers, whereas "3x" and "x.3" are not. Verify for yourself that this is the case by trying to trace each of these six identifiers through the syntax diagram in Figure 1.2.

The program section labeled block is essentially all of the Pascal program. The section preceding the block is often called the *program header.* A block has the structure shown in Figure 1.3.

Examples of statements are shown in Figure 1.4. Each line of Figure 1.4 is a separate statement, as indicated in the abstract syntax diagram in Figure 1.3, where semicolons separate all statements.

Since the Pascal language provides us with an enormous multiplicity of possible statements, we will discuss a specific statement and its syntax as it becomes useful.

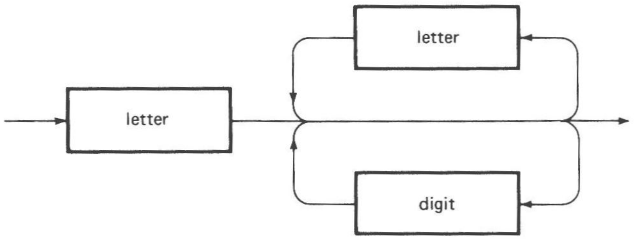


Figure 1.2

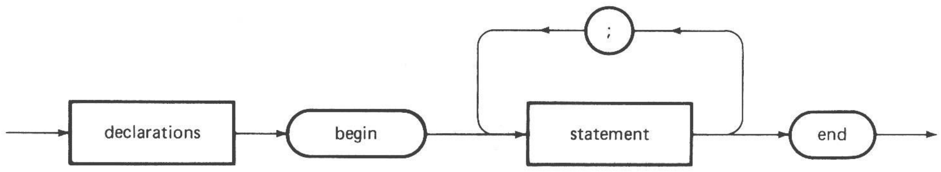


Figure 1.3

```
x:= 15;
while (y = 0) do search;
if (not tadpole) and (alive) then frog:= true;
if angle(x) = 1 then z:=5 else writeln('help!');
clearscreen;
z:= trunc(x+0.5) - y mod p + m*n;
```

Figure 1.4

```
program newton(output);
begin
    writeln('f = ma')
end.
```

Figure 1.5

Figure 1.5 is a legitimate Pascal program. Notice that the program name, an identifier, is "newton", that the file identifier is "output", that there is no declaration section, and that the single statement in the program is

writeln('f = ma')

This statement is a *procedure call.* It activates the standard procedure "writeln" (for write line), which writes things out. The items inside the parentheses, the *arguments* to "writeln", are what will be printed. In this case, the argument is a *string constant,* a group of characters enclosed in single quotation marks. Such quoted characters are taken literally. Therefore, when compiled and executed, this program will print

f = ma

and that is all it will do.

Exercises

1. By using the syntax diagram for identifiers, indicate which of the following are valid identifiers:

loopcount	xy	2x	2 + A
loop count	x y	x2	A − 2

```
program gaslaw(input, output);
begin
    writeln('ideal gas law:');
    writeln('PV = NRT')
end.
```

Figure 1.6

2. Consider the program shown in Figure 1.6. By using syntax diagrams, list these items:
   (a) The first keyword in the program
   (b) The first identifier (this will be the program name)
   (c) The file identifiers
   (d) The whole block
   (e) The operator that separates statements
   (f) The identifier naming the one procedure used in this program
   (g) The string constant representing the first piece of output printed

Answers

1. Valid identifiers: loopcount, xy, x2. None of the others can be properly traced through the syntax diagram.
2. (a) program
   (b) gaslaw
   (c) input, output
   (d)
```
begin
    writeln('ideal gas law:');
    writeln('PV = NRT')
end
```
   (e) ;
   (f) writeln
   (g) 'ideal gas law:'

WRITING OUTPUT

Consider the program shown in Figure 1.7. When run, this program will print the following:

```
newton's second law:
f = ma.
```

This example contains two items of interest. First, note that two apostrophes are used in a string constant when you want a single one to appear on output; second, after "writeln" has printed all its arguments, it prints a *carriage return/line feed* that forces the next item printed to appear on a new line. Pascal also has the

```
program newtonl(output);
begin
     writeln('newton''s second law:');
     writeln(' f = ma.')
end.
```

Figure 1.7

```
program manylines(output);
(* Quote-printing program *)
begin
    write('every body continues', ' in its state  ');
    writeln('of rest');
    write('or in uniform motion in a right line unless');
    writeln;
    writeln('it is compelled to change', ' that state by forces');
    writeln('impressed upon it. ');
    writeln;
    writeln('                    -- newton')
end. { manylines }
```

Figure 1.8

procedure "write", which allows the data printed by several different statements to appear on the same line.

Both "write" and "writeln" can take several arguments, and both will print all the arguments given them on the same line. Further, "writeln" can be used without any arguments. The program in Figure 1.8 will print the following six lines of text:

```
every body continues in its state of rest
or in uniform motion in a right line unless
it is compelled to change that state by forces
impressed upon it.

                    — —newton
```

The program includes a *comment* that explains the program's purpose. Comments are ignored by your machine when a program is compiled, but they help to explain a program's structure and function to the reader. Another reason to include comments is that when you return later to edit a section of your program, you may have forgotten your original rationale. This predicament is easily averted by way of good commenting habits. Comments are usually enclosed in symbols (* *) or in "braces" { }.

Exercises

1. Consider the program in Figure 1.9. How many lines of output will this program produce? What is the program name? For what phrase do you think the program name is an abbreviation?

```
program glop(input, output);

begin
    write('e=ir');
    write(' ');
    writeln('e=mc squared');
    writeln('div b = 0     epsi = hpsi')
end.
```

Figure 1.9

2. Write your own program to print the following text:

```
the test of
all
knowledge is
experiment!

—feynman
```

One blank line should be printed before the author's name. Enter your program on the computer, compile it, and run it.

Answers

1. There are two lines of output. The program name is "glop", which stands for great laws of physics. The great laws are Ohm's law, Einstein's equation, a Maxwell equation, and the Schrödinger equation.
2. The last three lines of such a program should be

```
  writeln;
  writeln('—feynman');
end.
```

IDENTIFIERS AND DECLARATIONS

There are two kinds of identifiers. *Standard identifiers* are automatically provided by Pascal. For example, "writeln" is a standard identifier that refers to a procedure that writes things. The other kinds of identifiers are invented by the user. You must declare them before they are used so that Pascal will know what each identifier names. This is done in the *declarations section* of a program, which has the form shown in Figure 1.10.

The simplest thing to identify is a constant. *Constants* are numbers, such as 5 or 3.1415926535, or strings, such as 'this string', which do not change. We can attach identifiers to constants with a *constant definition.* For example, we could define the identifier "e" to have the value 2.718281828. The constant declaration section of a program has the structure shown in Figure 1.11. The program "newton2" in Figure 1.12 has an effect identical to that of the program "newton" given previously.

One of the most useful things that can be identified is a variable. *Variables*, like constants, represent numbers, characters, and so on. Unlike constants, however, the *value* of a variable can be changed.

Imagine a variable as a box in which you place items to look at later. This box could be said to have a very specific shape, which is its type. The *type* determines what variables you can and cannot put into the box. For example, you can put the numbers 1, −3, 7/8 in variables declared to hold the type called real numbers.

All variables must be declared as such and assigned a specific type. This is done in the *variable declaration section* of a program illustrated in Figure 1.13. We will mainly be concerned with the two numerical types *integer* and *real*.

You can also declare your own customized types, procedures, and functions. These options come up later in the book.

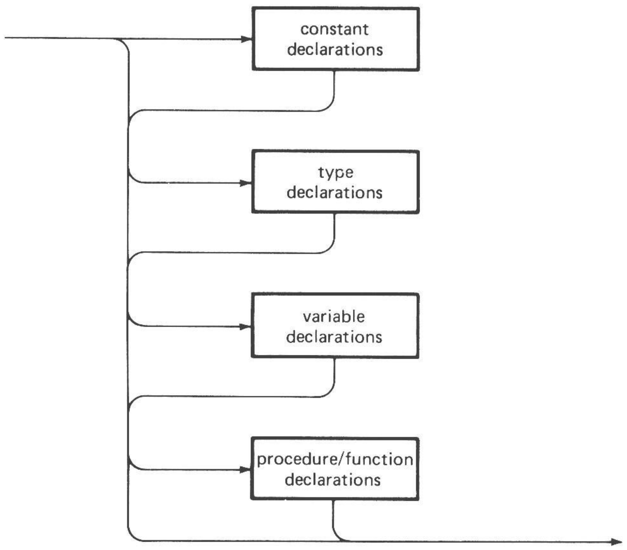


Figure 1.10

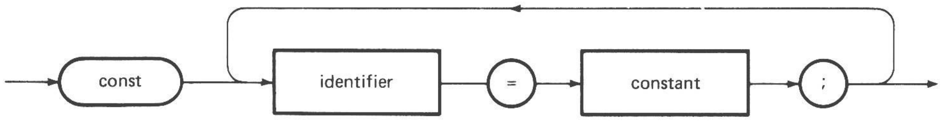


Figure 1.11

```
program newton2(output);
const
     law2 = 'f = ma ';
begin
     writeln(law2);
end.
```

Figure 1.12

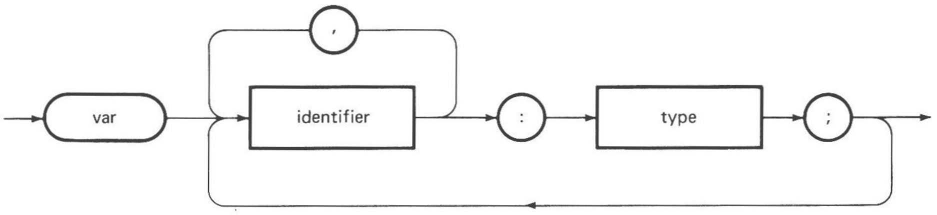


Figure 1.13

Exercises

1. Indicate which of the Pascal *types*—“integer”, “real”, “boolean”, “char”, or “illegal”—each of these values are:

   1.0 −99.7 6x true false maybe ‘h’ ‘65’
   1−0.2 99.012 −678

   For this and other problems you may wish to consult a Pascal reference (see Chapter 1 references).
2. For the program in Figure 1.14, list the following:
   (a) All standard identifiers
   (b) All keywords
   (c) All user-defined identifiers

Answers

1.

1.0	real
−99.7	real
6x	illegal

```
program exl(input, output);


(* print the area and circumference of a circle. *)
const
    pi = 3.14159265358979;
var
    radius, area: real;
begin
    radius := 2.35; (* set the variable 'radius' to a number. *)
    area := radius * radius * pi; (* calculate the area. *)
    writeln('radius =', radius);
    writeln('area =', area);
    writeln('circumference =', 2 * pi * radius)
end. { exl }
```

Figure 1.14

true	boolean
false	boolean
maybe	illegal
'h'	char
'65'	illegal
1−0.2	real
99.012	real
−678	integer

2. (a) Input, output, real, and writeln
(b) Program, const, var, begin, and end
(c) Exl, pi, radius, and area

CALCULATIONS IN PASCAL

In order to use variables, you must give them values. This is often done with an *assignment statement,* as shown in Figure 1.15. The section labeled "expression" is where you actually do calculations. The value calculated is then stored in the variable identified to the left of the ":=" operator. Thus,

```
x:=2
```

sets the variable "x" to the value 2.

Expressions can be extremely complicated. Consider the examples in Figure 1.16. Two things are immediately apparent. First, expressions use *operators* in

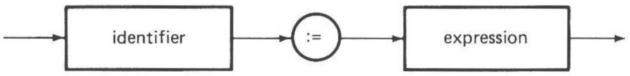


Figure 1.15

```
Expression          Value          Explanation

  15 + 2             17              ----
  15 * 2             30              ----
  15 / 2             7.5           real number division
 15 div 2             7            integer division
 15 mod 2             1            remainder after division
15 * 2 + 1           31              ----
15 * (2 + 1)         45              ----
15 / 1 + 1           16            same as (15 / 1) + 1
15 / (1 + 1)         7.5             ----
```

Figure 1.16