Pascal at Work & Play, Chapter 2:
Richard Forsyth;
Published by Chapman & Hall, London, 1982 (written 1981).

[Please cite as: Forsyth, R.S. (1982). *Pascal at work & play*. Chapman & Hall, London. 16-28.]

2. A Preview of Pascal

It is time to get down to business.

2.1 A simple example [BIRTHDAY]

Below is a Pascal program which, given a date, calculates the day of the week on which it falls. It works from 1582, when the Gregorian calendar was introduced, till 4903, when it will be one whole day out.

Do not worry if, at first reading, you cannot figure it out. In this chapter we have to expose you to a little bit of everything (some input/output, some arithmetic, some discussion of data types and so on) just to get started. All these topics will be treated in fuller detail in subsequent chapters, when you will be expected to understand them. For the moment it is sufficient that you gain an overall idea of what Pascal looks like in action.

```
program birthday;
(* calculates day of week given birthdate by zeller's congruence *)

const thisyear = 1982;

type days = 1..31;
  monthly = 1..12;

var d : days;
  m : monthly;
  year : 0..9999;
  ok : boolean;
  cent,dday,z : integer;

begin   (* main program *)
  (* initial message *)
  writeln('please give your birthday as three numbers:');
  writeln('for example, 2nd October 1948 as 2 10 1948');
  writeln('when you have finished give a date before 1582 or after 4902');

  repeat   (* main loop *)
    write('date of birth?');
    read(d,m,year);
    ok := (year > 1581) and (year < 4903);
    if ok then
      begin   (* use zeller's congruence formula *)
      if year <= thisyear then
        write ('you were born on a ')
      else
```

```
        write('you will be born on a ');
    if m<3 then
        m := m+10
    else
        m := m-2;

  (* months numbered from 1=march to 12=february *)
    if m>10 then
        year := year - 1;
        (* jan and feb considered to belong to previous year *)
    cent := year div 100;   (* century *)
    year := year mod 100;   (* year number in century *)
    z := trunc(2.6*m - 0.2);   (* magic formula *)
    dday := z + d + year + year div 4 + cent div 4 - 2*cent;
    dday := (dday + 777) mod 7;
        (* large multiple of 7 added to ensure nonnegative value *)
    case dday of
        0:  writeln('sunday');
        1:  writeln('monday');
        2:  writeln('tuesday');
        3:  writeln('wednesday');
        4:  writeln('thursday');
        5:  writeln('friday');
        6:  writeln('saturday')
      end;   (* of case*)
    end
  else
    writeln('you should live so long!');

  until not ok;

  writeln('have a nice day!');

end.
```

The program uses a method known as Zeller's congruence which states that the day of the week (DDAY) for any date in our era may be computed from the expression

dday = trunc(2.6*m-0.2)+d+y+trunc(y/4)+trunc(c/4)-2*c mod 7

where D is the day of the month, M the number of the month, Y the year in the century (0 to 99) and C the century number (Lee et al., 1978).

By TRUNC(X) we mean that X is evaluated and any fractional part ignored -- for example TRUNC(3.75) equals TRUNC(3.25) which equals 3. By Y/4 we mean Y divided by 4, and by X MOD Y we mean the remainder, or modulus, after dividing X by Y. Thus 22 MOD 7 = 1. The result DDAY will be from 0 to 6 and is interpreted such that 0 is equivalent to Sunday, 1 equivalent to Monday ... and 6 equivalent to Saturday.

Here is the output produced by a run of this program when it was executed on a DEC System-10, a large mainframe computer.

```
PLEASE GIVE YOUR BIRTHDAY AS THREE NUMBERS:
FOR EXAMPLE, 2ND OCTOBER 1948 AS 2 10 1948
WHEN YOU HAVE FINISHED GIVE A DATE BEFORE 1582 OR AFTER 4902

DATE OF BIRTH? 2 10 1948
YOU WERE BORN ON A SATURDAY
DATE OF BIRTH? 30 9 1975
YOU WERE BORN ON A TUESDAY
DATE OF BIRTH? 10 1 1954
YOU WERE BORN ON A SUNDAY
DATE OF BIRTH? 1 1 1900
YOU WERE BORN ON A MONDAY
DATE OF BIRTH? 29 2 1964
YOU WERE BORN ON A SATURDAY
DATE OF BIRTH? 7 10 1956
YOU WERE BORN ON A SUNDAY
DATE OF BIRTH? 25 12 1999
YOU WILL BE BORN ON A SATURDAY
DATE OF BIRTH? 1 1 1
YOU SHOULD LIVE SO LONG!
HAVE A NICE DAY!
```

Note first of all that this is an example of interactive computer use. The user types something at the keyboard, and the computer responds by printing its reply. Most large computers (including the DEC System-10) and all microcomputers can work in this manner nowadays.

This is a case of a program built around a 'magic formula' taken from a book. Notice, however, that the formula itself

```
z := trunc(2.6*M-0.2);
dday := z + d + year + year div 4 + cent div 4 - 2*cent;
dday := (dday + 777) mod 7;
```

is tucked away in three lines of a 66-line program, and could have been squeezed into one line if we had wanted. There is much more to writing a program than knowing how to calculate the answer.

2.1.1 Behind the scenes

A program like this, though small by the standards of 'real life' computing, is liable to give the novice programmer conceptual indigestion when presented as a fait accompli. But programs do not spring ready-made like Athena from Zeus's forehead; so I will attempt to give a slow-motion action-replay of the thought processes that led to its creation. This should dispel the aura of mystery and show that program design is not a conjuring trick, merely the repeated application of a few simple principles.

I began -- let us be honest about it -- with the need for an interesting but not too complex example to put in my book. While reading about something else my eye was caught by a description of Zeller's congruence method. It fitted the bill because a computer could do it easily but a person would take longer to work out the answer than it was worth -- and that person, speaking for myself, would probably make a mistake in the process. So I arrived at the following ground plan.

```
program birthday;

(* declare constants, datatypes and variables *)

begin

  (* give user some instructions *)

  repeat
    (* obtain the date in question *)
    (* transform to suitable form *)
    (* apply zeller's congruence *)
    (* print the corresponding week-day *)
  until not ok;   (* no more to do *)

end.
```

This skeleton is nearly, but not quite, a valid Pascal program. It is invalid because the variable OK which will be used to indicate that the given date cannot be processed, has not been declared. Even if OK had been declared, the program would still do nothing. That does not, however, mean that it is useless, far from it. It is a framework on which to build.

Let us begin by considering what '(*' and '*)' are for. These are the Pascal comment symbols. Everything written between (* and the next *) is treated as commentary and ignored by the Pascal compiler. In other words it represents a long-winded way of doing absolutely nothing. All comments can be removed from a Pascal program without affecting its performance. Why then do I maintain that this feature is one of the most important in the language?

There are two reasons: firstly, comments, judiciously worded, make a program easier to read and understand; secondly, and more important, they are indispensable to the program development process. During program-writing they correspond to the Open University's thought-bubble symbol described in Section 1.3. They permit the application of 'stepwise refinement' in which an initially skeletal program structure is fleshed out in greater and greater detail. (The Pascal standard actually designates the curly braces '{' and '}' to enclose comments, but many printers do not have these characters so (* and *) are recognized alternatives.) A comment may be placed anywhere in a program where a blank space is permitted.

Let us briefly consider the other features of Pascal introduced so far.

The outline begins with

```
program birthday;
```

which identifies the program by giving it a name, BIRTHDAY. The semicolon (;) is important too. It is used to separate statements. We will explain the rules of its use in Section 2.3.

Next will come the declarations of data to be used by the program. We have postponed a decision on this until we know what will be required, i.e. till after the processing is specified.

The processing lies between the first BEGIN and the last END. Note that the final END is always followed by a full stop (period) to terminate the program text.

Finally, everything between REPEAT and UNTIL is performed repeatedly until the 'condition' after UNTIL is satisfied. The condition is the expression between UNTIL and the semicolon which must be true or false, in this case NOT OK.

We can now proceed with the second phase of our stepwise refinement, leaving the data declarations till last.

The comment

```
(* give user some instructions *)
```

is replaced by

```
writeln('please give your birthday as three numbers:');
writeln('for example, 2nd October 1948 as 2 10 1948');
writeln('when you have finished give a date before 1582 or after 4902');
```

which should be enough to tell the user what the computer expects. For the time being all you need to know is that WRITELN('XYZ') will cause XYZ to appear on the default output channel (normally the user's terminal) immediately followed by a new line, while WRITE('ABC') will cause ABC to be printed without moving to a new line.

For

```
(* obtain the date in question *)
```

we substitute

```
write('date of birth? ');
read(d,m,year);
```

which will print

```
DATE OF BIRTH?
```

on the terminal and then wait for the user to supply three numbers, separated by one or more spaces or new lines, for D, M and YEAR respectively. D, M and YEAR are called variables since their values can change during a computation (see Section 3.3).

We now come to an oversight in the original design: there is no provision for handling bad input. When a program reads data from a user at a console there is always the possibility of a mistake. In this case we want particularly to ensure that the year is within the range for which a meaningful answer can be given. So we perform the assignment

```
ok := (year > 1581) and (year < 4903)
```

and make the rest of the computation conditional on OK, which will be either TRUE or FALSE. Our revised schema for the main cycle is

```
repeat
```

```
 write('date of birth? ');
 read(d,m,year);
 ok := (year > 1581) and year < 4903);
 if ok then
   (* transform to suitable form *)
   (* apply zeller's congruence *)
   (* print the corresponding weekday *)
 else
   writeln('you should live so long!');
 until not ok;
```

which gives some measure of protection against spurious results.

I put in this slight revision not to make things more confusing but to reflect the history of the program's creation more faithfully. Even with a short program like this one often has second thoughts.

Notice that, although the user will not be able to give a day number outside the range 1 to 31 or a month number outside 1 to 12 because of Pascal's type-checking, the program is still not entirely foolproof: dates such as 31 9 1975 or 30 2 1980 will still be accepted.

The description

```
(* transform to suitable form *)
```

is turned into Pascal statements that adjust the month number so that months run from March (1) to February (12) and alter YEAR so that January and February are considered part of YEAR-1.

Look back now at the program and see if you can see for yourself how

```
(* apply zeller's congruence *)
(* print the corresponding weekday *)
```

were expanded. You should be aware that DIV is the integer division operator. YEAR DIV 4 has the same effect as TRUNC(YEAR/4) -- i.e. the remainder is lost. MOD, as already mentioned, gives the remainder so that if YEAR starts as 1999 the statements

```
cent := year div 100;
year := year mod 100;
```

leave CENT with the value 19 and YEAR with 99. You also need to know that the CASE statement selects one of many alternatives. In the example program the selector is DDAY so that if DDAY=5 the single statement labelled with 5: is executed (and none of the others between CASE and END). When DDAY=5, for instance, the statement

```
writeln('friday');
```

is the only one chosen.

All that is left is to turn

```
(* declare constants, datatypes and variables *)
```

into valid Pascal declarations.

The declaration

```
const thisyear = 1982;
```

introduces a symbolic constant. The name THISYEAR will have a fixed value of 1982. The reason I did not just write 1982 in the program wherever THISYEAR appears is to make it obvious what change would be needed to run the program in, say, 1984. Also, if THISYEAR had been used in many places, only one change, such as

```
const thisyear = 1984;
```

would be enough to change all occurrences.

The declaration of new data types as in

```
type days = 1..31;
  monthly = 1..12;
```

is a particularly interesting feature of Pascal. The types DAYS and MONTHLY now become 'subranges' of the integers. Legal values for variables of these types are restricted to lie within the subrange specified. This is most valuable for error prevention. Other typing facilities are described in Section 3.4.

Finally,

```
var d : days;
  m : monthly;
  year : 0..9999;
  ok : boolean;
  cent,dday,z : integer;
```

sets up the required variables. YEAR will be able to hold values from 0 to 9999. OK is a BOOLEAN variable and can have one of the values TRUE or FALSE. CENT, DDAY and Z are integers: they can take any whole-number values that the computer is capable of representing, positive or negative.

The fact that a Pascal programmer can declare new types and can restrict variables to having only values that are likely to be reasonable for the problem in hand contributes greatly to program robustness. Many older languages (e.g. Basic and Fortran) do not offer this ability. Yet it is seldom, for example, that the full range of integers is really needed, and by reducing the number of allowable values the chances of an error going undetected are reduced. In this program a 32nd day or a 13th month would be rejected at once.

I have elaborated on this example at some length because I believe that sooner or later (preferably sooner) the would-be programmer must encounter some valid programs. You cannot learn to swim without getting your feet wet. I also wanted to demonstrate that programs are not made by black magic but by the methodical application of commonsense principles.

2.2 Syntax diagrams

You have seen an example of Pascal in use. Now we define some of its rules. To define the grammar of the language we employ the method popularized by Wirth -- 'syntax diagrams' (Wirth, 1973). A syntax diagram is like a flowchart except that it describes data format rather than processing actions.

Figure 2.1 is a set of syntax diagrams outlining the rules for a subset of American male names. There are two kinds of boxes, round and rectangular. The round boxes enclose terminal symbols such as 'Richard'. These stand for themselves and appear as they are. The square or rectangular boxes refer to subsidiary syntax diagrams defined elsewhere. Thus 'Forename' appears as a constituent of 'Style' and has its own definition, in terms of another syntax diagram.

To generate an allowable construction from a syntax diagram you follow the arrows from box to box until you reach the exit. Where there is a branch you can take either fork. Where there is a reference to another syntax diagram you enter it, follow it through, and then return to where you left off in the original diagram. We will employ syntax diagrams to clarify the rules of Pascal in this book from now on. A complete description of Pascal syntax is given in Appendix D.

[Figure 2.1 Sample syntax diagrams.]

According to the above example the following names are valid:

President John F Kennedy
Mr R M Nixon III
Professor George George George George Washington
Mr Henry Ford

The following names could not have been produced from the diagrams in Fig. 2.1.

Mr Ford
Abraham Lincoln
President H Kissinger
Ronald Reagan

You should try to understand in what respect each is invalid before reading on.

Mr Ford is invalid because Style includes at least one Forename or Initial; Abraham Lincoln is badly formed because it has no Title; President H Kissinger is invalid because H is not one of the instances of Initial; and Ronald Reagan is wrong in almost every respect.

Just to ensure you have grasped all this let us construct one more name from the syntax diagrams.

We enter Name and come across Title. So we enter Title and take a right fork, giving

Dr

as output. Having completed Title we return to Name and move on to Style. Entering Style we branch right and hit Forename, so we enter Forename and, to cut a long story short, pick

John

and return to Style.

Passing along from Forename we fork left and thus dive into Initial. Here we choose

J

and exit. Now we have finished Style and can return to Name where we proceed to the box marked Surname. In Surname we produce

Johnson

and return to Name. Leaving the Surname box we bear left and go through Extra Bit, which yields

Jnr

before finishing Name altogether. We have thus created

Dr John J Johnson Jnr

as a result of our travels. (Note that though Dr J J Johnson and Dr John John Johnson are permitted Dr J John Johnson is not.)

2.3 Program structure
Now we can use a syntax diagram to exhibit the structure of a Pascal program in an unambiguous way (Fig. 2.2). We are not yet ready to define Declaration or Statement yet. Declarations are covered more fully in Chapter 3 and statements beginning in Chapter 4. However we can define Identifier as shown in Fig. 2.3 which states that an identifier is a letter optionally followed by any number of letters or digits. (Actually Pascal does not guarantee to distinguish between long identifiers that are the same in their first eight characters, so we will accept eight characters as a maximum length for identifiers in this book.)

[Figure 2.2 Program syntax]

What do these diagrams mean?

They state that a program has a name, possibly followed by a bracketted list of identifiers (which in fact designate files) followed by a semicolon. This is the program heading. Next come the declarations, which are for introducing data names and so on, followed by a BEGIN, one or more statements separated by semicolons, and an END. (The full stop is part of the definition.)

[Figure 2.3 Identifier syntax]

The declarations define the data characteristics; the statements accomplish the processing.

2.3.1 Reserved words and other symbols
A Pascal program is composed of characters. The letters, digits and most of the punctuation marks need no further explanation, but there are also various strings of characters designated as single entities. These letter sequences are sometimes called 'reserved words' -- reserved in that the programmer is not free to re-define their meanings.

For example, though BEGIN consists of five letters and END contains three they are treated as units by the Pascal compiler. A complete list of reserved words appears in Appendix B.

Certain other compound symbols such as

:= <= >= <> (* *)

are also treated as indivisible items. Blank spaces are not permitted between the characters of a compound symbol or reserved word.

In most other respects the format of a Pascal program is very flexible. The end of a line does not terminate a statement. Pascal statements are separated by semicolons. Consequently a line may contain several statements, or one statement may extend over several lines. Blank spaces and new lines may be inserted to make the program look pleasing to the eye, and to show its logical structure on the page. In this book we follow a consistent pattern of indentation which shows how the statement groups are related. This freedom from rigid card or margin boundaries is a welcome improvement over, for instance, Cobol and Fortran.

2.4 A word in edgeways

Among the tribulations of the novice programmer, which make learning a programming language seem child's play in comparison (it is!) are: learning to type, learning to use an editor program, finding your way about the system.

Keyboards vary quite a lot in the placement of punctuation and special characters, though they almost all stick to the traditional QWERTY layout for the alphabet -- which, incidentally, was devised to slow down typists in the 1880s when the machinery could not keep pace with nimble fingers. Until the day of widespread voice input we must all put up with it.

Editors are a necessary evil because they let you create (and amend) a file of text containing your program, which can then be presented to the compiler for translation into machine instructions; but the less said about some of them, the better.

If you use a microcomputer it will undoubtedly have quirks of its own (e.g. you may find a program will run perfectly from drive A but fail to load from drive B). If you use a large computer you will have to go through various procedures before you can even become an authorized user, and then there is the problem of 'logging in' -- not a trivial task on some of today's systems.

All these mundane difficulties just make it slightly harder to get useful results from the machine. You may be put off by them, especially if you had an image of programmers as creatures of pure thought who would lock themselves in a darkened room to indulge in abstruse mental gymnastics before bursting out with a cry of 'Eureka!'.

This book cannot help you much with such practicalities. They are too various. But do not despair. Eventually we all learn to type at an acceptable rate, even if only with two fingers and the occasional thumb. It is not beyond the wit of ordinary mortals to master the intricacies of most text-editors either (though they often seem designed to make it so). And you can even learn to live with a micro that only likes its floppy discs 'sunny side up' or a mainframe that forces you to go through a complicated ritual in order to 'log in'.

Perseverance is essential. Once you have surmounted these barriers you will find that the computer is useful after all.