A First Look At PASCAL.

1 Coding.

We now must learn how to turn Pseudocode into real program text, often referred to as code.

All programming languages have some basic structure. This means that the syntax has to be obeyed carefully. You already saw some of that when writing Pseudocode.

2 Programming Languages.

Like every language, not just computer languages, there is a meaning attached to every word and grammatical construct.

2.1 The Vocabulary

Typically computer languages only have a few words with any meaning attached to them.

Most computer languages have less than 100 words with any predefined meaning.

Compare this with English which has many more, the average pocket dictionary contains close to 15,000 words, the Oxford English Dictionary (2nd Ed.) has approx 500,000.

There are only a few keywords and their meanings are simple. It’s not like English where you may find yourself wondering what the word quondam means and how it differs from erstwhile.

2.2 The Grammar.

There are also only a handful of grammatical constructs in computer languages.

This is enough for us to exploit our computer to the maximum. We can string together statements built using predefined words, known as keywords, to do more complex tasks.

We can also assign each of these more complicated tasks a name which we can use later.

e.g. We could write a small Pascal program called AskAndRecord which we could use whenever we wish to ask a question and record the answer.

2.3 You can do anything...

Programming languages are designed to be ‘‘Turing complete’’. This means that if there is a problem that can be solved on a computer we can write a program, in the programming language to solve it.

2.4 Mastering a programming language.

Mastering a programming language is a matter of understanding these keywords and its grammar.

With a simple language like Pascal you can solve all the worlds computer problems!

Good programming also requires technique, this only comes from writing programs and that takes time.

3 Some Simple Programs.

So now let’s look at some Pascal!

3.1 The Simplest Program.

Here is the simplest Pascal program.

```
PROGRAM Null;
BEGIN
END.
```

This is the smallest legal Pascal program. It has:

A program header. This is made up of three parts,

1. The keyword PROGRAM followed by,
2. The program identifier, Null.
3. The semicolon.

together they serve to give the program a name. This program is called Null.

A statement part. The keywords BEGIN and END are delimiters, much like parentheses in English. They mark the BEGINning and END of the instructions, or statements in a program.

What are the keywords in the above program?

What does this program do? The answer is nothing! Roughly speaking, what we’ve done here is created a new program, identified by Null, to which we attached the meaning - nothing.

3.2 Introducing the write() statement.

Here is another simple Pascal program.
1. The program header identifies this program, i.e., the set of actions taken between the BEGIN and END keywords, with the name HelloWorld.

2. This program actually does something. The line
   \[\text{write('Hello World')}\]
writes the words between the quotation marks
   \[\text{Hello World}\]
to the computer screen.

3. Write is not a keyword! So what is it? Write is a \textit{predefined procedure}, or a standard procedure. It is provided for you to use freely with every Pascal system but is not part of the Pascal language. It is called a standard procedure because all versions of Pascal are required to provide it. Write is a solution to the problem of "writing to the screen". It is really another program which your program uses. It may be written in Pascal, and will perform all them tasks needed to get the quoted part onto the screen. It is probably written in Pascal as well. There are lots of standard procedures in Pascal. They are provided for your convenience. You must remember at all times that they are part of the Pascal system but NOT part of the Pascal language.

4. The word \textit{output} in parentheses is special. It is used to tell the Pascal compiler that the program will be used to output something onto the screen. If we had input we'd need to put the word input in its place. If we had both input and output we'd need to write both.

3.3 Some subtleties about write().

We can make our example a little more complicated. Try this

\[\text{PROGRAM HelloWorld(output);}\]
\[\begin{align*}
\text{BEGIN} & \quad \text{write('Hello World');} \\
\text{END.} & \quad \text{write('Hello World');}
\end{align*}\]

This will produce

\[\text{Hello WorldHello World}\]

The computer performs all the instructions between the BEGIN END delimiters in sequence.

If we want a space we need to put it in. Either as

\[\text{PROGRAM HelloWorld(output);}\]
\[\begin{align*}
\text{BEGIN} & \quad \text{write('Hello World');} \\
\text{END.} & \quad \text{write(' Hello World');}
\end{align*}\]

or

\[\text{PROGRAM HelloWorld(output);}\]
\[\begin{align*}
\text{BEGIN} & \quad \text{write('Hello World');} \\
\text{END.} & \quad \text{write(' Hello World');}
\end{align*}\]

both of these will say

\[\text{Hello World Hello World}\]

NOTE: The ';' is used to separate statements!

3.4 The writeln() statement.

\texttt{WRITELN} is also a standard procedure, it prints the desired output and then goes to the next line.

\[\text{PROGRAM HelloWorld(output);}\]
\[\begin{align*}
\text{BEGIN} & \quad \text{writeln('line one');} \\
\text{END.} & \quad \text{writeln('line two')}
\end{align*}\]

will print

\[\begin{align*}
\text{line one} \\
\text{line two}
\end{align*}\]

3.5 We can use both write() and writeln() together.

We can mix write and writeln freely too

\[\text{PROGRAM HelloWorld(output);}\]
\[\begin{align*}
\text{BEGIN} & \quad \text{write('line one');} \\
\text{END.} & \quad \text{write(' line two');} \\
\text{writeln('still on line one');} \\
\text{writeln('still on line two')}
\end{align*}\]

What will this produce?
3.6 Something to try at home.

Try this on paper tonight, that is you don't have to hand it in and you don't have to enter it into the computer just yet.

Can you write a program to print out the program Null? ie.

```pascal
PROGRAM Null;
BEGIN
END.
```

should appear on your screen.

How many write and/or writeln statements do you need to use, ie what's the least amount you can get away with?

How many write and/or writeln statements do you need to use maximally?

Write and test the program using the Pascal compiler. (You don't have to do this last part but it helps to get comfortable with the system you'll be using before the real assignments start!)

4 Variables and Values.

To make our programs do more sophisticated tasks we need to introduce the notion of a variable.

Consider the following (incomplete) code, to conduct the survey we discussed in the section on Pseudocode. How do we create the boxes to hold the counts?

We use a variable. A variable is a place in memory that we use to keep a value. It is called a variable because its value can be changed during the course of a program.

```pascal
PROGRAM Survey;
(* Declare variables to store the values of NEWS *)
VAR North, East, West, South: ???;
BEGIN
(* Initialize the counters to zero *)
North := 0;
East := 0;
West := 0;
South := 0;
(* rest of program follows *)
blah blah...
END.
```

A variable declaration has the following form.

```pascal
VAR Name, ... :Type;
```

It is always placed after the program name and before the BEGIN END statement. The name North now can hold a value.

4.1 Using Variables and Assignment Statements.

We assign the value zero to the variable north as follows.

```pascal
North := 0;
```

The line above is known as an assignment statement. The proper way to understand it is to say “the value of North is zero” or “North gets zero”.

Now whenever we need to know the value in the variable North we just use North. Suppose we wanted to know the value of the North and East we could just use the value of

```pascal
North + East;
```

in the BEGIN END statement that follows the VARIABLE declaration.

4.2 Naming Variables with Identifiers.

The names of different variables are known as identifiers, you can think of them as the names on the boxes, or more correctly, the name of the memory location where the variable is stored. In our example above North, East, South and West are all variables. The word North is the identifier for the location where the value of North is stored.

There are some rules here too.

1. An identifier must start with a character and can't have any blanks in it. It may have digits after the first character.

2. The compiler is case insensitive. ie the identifiers length, Length and LENGTH are all the same identifier.

These are examples of legal identifiers:

Length, Width, North, North2South, FromHereToEternity,

these are examples of illegal identifiers

2Bad, Once upon a time, A*ForGoodBehaviour No,’sAllowed.
5 Reserved Identifiers.

Some identifiers are reserved, that is a programmer can’t use them in his program except for their predefined meaning in the Pascal language. Trying to name a variable with one of the program identifiers is an error. e.g.

\[ \text{VAR CONST} = 2.0; (* \text{You can’t have a variable called CONST} *) \]

Reserved words are also known as keywords.

5.1 Pascals Keywords.

The following words are reserved:

- and
- array
- begin
- case
- const
- div
- do
- downto
- else
- end
- file
- forward
- for
- function
- goto
- if
- in
- of
- or
- packed
- procedure
- program
- record
- repeat
- set
- then
- type
- to
- until
- var
- while
- with

6 Types.

The type of an expression tells us the values it can denote and the operations that can be applied to it.

When a computer is told to add two variables we hope that the two variables are numbers. Does this make sense?

\[ 3 + ‘a’ \]

or

\[ ‘a’ + ’b’ \]

where ‘a’ and ’b’ are the letters ‘a’ and ’b’ and have no other meaning attached to them.

We know that

\[ 3 + 4 \]

is a reasonable operation to expect the computer to perform.

6.1 The type system, or “How types enforce meaning”.

To avoid meaningless statements like \( 3 + ‘a’ \) we introduce the idea of a TYPE system. Types prevent us from carrying out nonsensical computation.

All variables must have a Type. This has to do with how computers are built. The Pascal compiler needs to know how to translate the source program into executable machine language.


We get to decide what kind of value North is in our program design. Of course we should choose from the tools available to us. Pascal provides us with the following basic classes of values.

- Character
- Integer
- Real
- Boolean

6.2 How does the Type system work?

The Pascal standard insists that every variable and piece of data must have a type associated with it.

When the compiler translates your program to machine language it uses this type information to figure out whether the instruction issued will result in a meaningful result.

If there are instructions that don’t result in a meaningful result the compiler will report a type error. All programs that compile will have, by virtue of our type system, no type errors, and are called type safe programs.

Since Pascal only permits type safe programs it is called a strongly typed language.

There are programming languages that are not strongly typed, any programming language that isn’t strongly typed is known as a weakly typed language. In these languages the line \( 3 + ‘a’ \) will be accepted by the compiler, but when it is finally run the program may crash, or even worse, give you a wrong answer.

7 Pascal’s Basic Types.

7.1 INTEGERs.

Integers are all numbers of the form

\[ \ldots, -3, -2, -1, 0, 1, 2, 3, \ldots \]

On computers however numbers can’t be infinite, instead we have a range of numbers, that we can use legally. So our integer types look like

MININT, ..., -3, -2, -1, 0, 1, 2, 3, ... MAXINT

The numbers MAXINT and MININT are defined by the system you are on. It may vary from compiler to compiler too.
In Turbo Pascal for example MAXINT is 32767 and MININT is -32768.

Integers can be written using only the numbers 0,1,2,3,4,5,6,7,8,9 and may optionally have a '+' or a '-' sign at the beginning. They may not contain any other characters, and they may not contain blanks.

These are legal integer declarations.

1, 2, 3, 4, 0004953687, +9846753, -6745, 0

These are illegal

2.3, 3-456, 9 45, 5$9 3+445, 76-, 45+, 12,543

8 Back to V ARIables for a bit.

Look back to the survey program, we now can fill in the type of the variable in the variable declaration line.

PROGRAM Survey;
(* This will calculate the area of a number *)
VAR North, East, West, South: integer; (* NEWS are Integers *)
BEGIN
(* Initialize the counters to zero *)
North := 0; (* We now give the variables a value *)
East := 0; (* of 0, which is legal because 0 is *)
West := 0; (* an Integer. *)
South := 0;
...
END.

This would have been illegal

South := 0.0;

As an aside here remember that the program is run from top to bottom. ie North is assigned first, then East, then West and lastly South. If we’d had this sequence of events

North := 0;
South := North;

Then South would get the value of North, ie 0.

8.1 Initializing variables correctly.

What about

South := North;
North := 0;

What is South equal to now? What is North equal to?

The answer is we don’t know. South could be anything. This is because the compiler finds an unused place in memory and decides that South will be stored there. It doesn’t know what value to put in that part of memory.

In more simple terms, the compiler finds a box for the value of south to be stored in. It doesn’t check to see what’s in the box! South got the value of whatever was in the North box. We didn’t know what was in the North box. Then we assign the value zero to the North box and we are left with South being anything but North being zero. At the first line of the program we say that the value of North was undefined, so too is the value of South.

A variable that has been set to a know value before it’s first usage is known as an initialized variable. The process of giving the variable a value for the first time is known as initialization.

A variable that is used before it has been initialized is known as an uninitialized variable.

8.2 A program with uncertain behaviour.

PROGRAM PrintZero(output);
VAR theValueOfZero:integer;
BEGIN
write(‘The value of zero is ‘);
writeln(theValueOfZero)
END.

This could produce the following output

The value of zero is 2.7346387 or

The value of zero is A

8.3 Initialize! Initialize! Initialize!

Write this backwards on your forehead so you learn it when you brush your teeth in the morning!

To make sure our program works we follow the rule of thumb below.

MAKE SURE YOU INITIALIZE A VARIABLE’S VALUE BEFORE YOU USE IT IN YOUR PROGRAM.
This is how you should write your program.

```pascal
PROGRAM PrintZero(output);
  VAR theValueOfZero:integer; (* A memory location to store zero *)
BEGIN
  theValueOfZero := 0; (* The initialization *)
  write('The value of zero is '); writeln(theValueOfZero)
END.
```

This will produce

```
The value of zero is 0
```

9  ... and now more Pascal Types.

Some other basic Pascal types are

9.1 Character.

A variable of type Character can any single character in the range a-z and A-Z and also include the character '$'se

These are legal characters,

```
A, B, a, $
```

These are illegal,

```
ab, $a, fred
```

9.2 Real.

A variable of the type Real can be a series of digits, but it must start with a digit and contain only one decimal point in it. It can of course be preceded by a plus or minus sign.

These are legal reals

```
0.2, 0.0, 1.0, -1234.435768
```

These are illegal reals

```
.200, 123, 1,234.0, 1.0.0.2
```

Now let’s write a simple program to use Reals.

```pascal
PROGRAM CircularArea(output);
  VAR Pi, Radius, Area: Real;
BEGIN
  Pi := 3.14159;
  Radius := 3.0;
  Area := Pi * Radius * Radius
  write('The area of a circle of radius ',Radius);
  writeln(' is ', Area, '.')
END.
```

When we run this program we get

```plaintext
The area of a circle of radius 3.00000000E+00 is 2.82345434E+01.
```

There are a few things we need to notice here.

- All the variables were initialized before use, that’s why we get a predictable result. In the line

```
Area := Pi * Radius * Radius
```

the first use of the variable Area is actually an initialization.

- The write() procedure is being used in a new way. Here we have a string of characters ‘The area of a circle of radius ’ followed by Radius We can put as much as we need to in a single write, or writeln procedure. All we need to do is separate the different elements being printed out by a comma. So

```
write('The area of a circle of radius ',Radius);
```

first writes “The area...” and then the value of the variable Radius.

- The values printed out by write(”) for Radius are in exponential form. The value of the area is 28.2345434. but in exponential form this is written 2.82345434E+01. We will see how to write out real numbers in fixed point notation later.

9.3 Boolean

A variable of Boolean type can have one of two values True or False.

We can write, due to case insensitivity of the Pascal compilers, the booleans as TRUE, True, true, TrUe ... and similarly false, FALSE, False, faiSE.

10 Operators.

Now that we know what types are we need to know what we can do with data of different types. We use operators to perform the work that we need to do in our programs. Look at the following statement
Value := 3 + 4;

Here ‘+’ is the operator, ‘3’ and ‘4’ are the operands, the phrase “3 + 4” is known as an expression, with the assignment the whole line is a statement.

Different types have different operators. For example we don’t have a Pascal operator ‘-’ to subtract two characters, ie ‘a’ - ‘b’ makes no sense so we don’t allow a programmer to try to do it. But we do have a subtraction operator for Integers and for Real.

### 10.1 Real Operators

For Reals we can use the following operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>3.0 + 2.0</td>
<td>5.0</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>3.0 -</td>
<td></td>
<td>Subtraction</td>
</tr>
<tr>
<td>2.0</td>
<td>1.0</td>
<td>1.0</td>
<td>Multiplication</td>
</tr>
<tr>
<td>*</td>
<td>3.0 * 2.0</td>
<td>6.0</td>
<td>Division</td>
</tr>
<tr>
<td>/</td>
<td>3.0 / 2.0</td>
<td>0.67</td>
<td></td>
</tr>
</tbody>
</table>

### 10.2 Integer Operators

For Integers we have the following operators:

<table>
<thead>
<tr>
<th>Operator</th>
<th>Usage</th>
<th>Result</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>3 + 2</td>
<td>5</td>
<td>Addition</td>
</tr>
<tr>
<td>-</td>
<td>3 -</td>
<td></td>
<td>Subtraction</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>1</td>
<td>Multiplication</td>
</tr>
<tr>
<td>MOD</td>
<td>3 MOD 2</td>
<td>1</td>
<td>Remainder</td>
</tr>
<tr>
<td>DIV</td>
<td>3 DIV 2</td>
<td>1</td>
<td>Quotient</td>
</tr>
</tbody>
</table>

### 10.3 Division.

Before we move on let’s look at division for Reals and Integers more closely. For reals we can use the ‘/’ to divide. This gives us familiar results. 3.0 / 4.0 = 0.75.

For Integers we have MOD and DIV. They differ from ‘/’ in that they produce the remainder and the quotient. Go back to your baby math and recall that an improper fraction, say 7/2 is actually 3 + 1/2. Here 3 is the quotient and 1/2 is the remainder. So the result of doing 7 DIV 2 is 2 and the result of doing 7 MOD 2 is 1.

### 12 Operator Precedence.

How do we evaluate the following expression.

\[ Y_1 - Y_2 / X_1 - X_2 * 3.0 + 4.1 \]

To most people familiar with the equation above we’d hope that \( Y_1 - Y_2 \) is divided by \( X_1 - X_2 \) and then multiplied by 3.0 and added to 4.1.

What in fact happens is that \( Y_1 \) is subtracted from \( Y_2 / X_2 \) which is then subtracted by \( X_2 \) and then added to 4.1. This is because of operator precedence.

Pascal, like almost all programming languages, attaches a higher priority to multiplication and division, than it does to addition and subtraction. This is known as operator precedence.

### 12.1 Using Operator Precedence.

This is how the computer will evaluate the expression above. Read the expression from left to right, performing all the multiplications and divisions first. So our original expression

\[ Y_1 - Y_2 / X_1 - X_2 * 3.0 + 4.1 \]

becomes

\[ Y_1 - \frac{Y_2}{X_1} - \frac{X_2 \times 3.0}{4.1} \]

now go back and evaluate all the additions and subtractions to get

\[ \left[ \left[ \left[ Y_1 - \left[ \frac{Y_2}{X_1} \right] - \frac{X_2 \times 3.0}{4.1} \right] \right] \right] \].
12.2 ... but we can use ( and ) to make programming easier!

As in mathematics we can place brackets around an expression to make sure that those operations within the brackets are performed first.

So the original equation would be written

\[(Y1 - Y2) / (X1 - X2) * 3.0 + 4.1\]

This forces the Pascal language to evaluate \(Y1 - Y2\) and \(X1 - X2\) first, then perform the divisions and multiplications, and finally add 4.1.

Do you know what the equation above is?

13 Mixed Mode.

When we calculate the value of an expression we usually end up assigning it, via an assignment statement, to a variable. eg

\[\text{Value} := (Y1 - Y2) / (X1 - X2) * 3.0 + 4.1\]

The type of value on the left hand side of the expression determines the type of the expression on the right hand side.

In the expression above the right hand side must be of type Real. This is because we’re using the “/” operator.

If we had \(2/3\) we should get an error, but Pascal is allows this by implicitly coercing the value of \(2/3\) into \(2.0/3.0\). An expression such as \((2/3) + 4.0\) is said to be written in mixed mode.

Note: In general it is UNSAFE to write programs in mixed mode. It is a BAD practice and should be avoided. Nonetheless in some cases it’s passable and you won’t get yelled at for using it.

14 The readln() statement.

It should come as no surprise to you that we have a way of getting data into a program from the outside world. This comes in the form of the READLN statement.

Look at the following program,

```
PROGRAM ReadAnInteger(input, output);
VAR Value1, Value2:Integer;
BEGIN
  writeln('Please Key in a pair of integers');
  readln(Value1,Value2);
END.
```

1. We need to add the word input to the program heading to tell the compiler to expect input from the keyboard.
2. We created variables called Value1 and Value2 of type Integer. This is to store the values that we input at the keyboard.
3. The first writeln statement is used to prompt the user for input. Without it we wouldn’t know when to input the value.

The program will say,

Please key in a pair of integers and then wait for the input.

4. readln() is the procedure used to read in from the command line. Now we enter the values. We could type, 2 3 [CR]

There are two values. They are separated by spaces. The [CR], or carriage return, tell the program that you’re done inputting two values. Forgetting the [CR] results in your program waiting for input.

15 CORRECTING SYNTAX ERRORS, (Getting your code to compile.)

From what you’ve seen so far writing a program is easy! The only thing you have to do is remember what instruction to use and you can write some simple programs.

You’re going to make mistakes. You’ll probably write

```
PROGRAM Volume(input, output);
(* Investigates reading *)
VAR Width: integer;
    Cost, Price:Real;
BEGIN
  writeln('Type in Length, Width and Price');
  read(Length);
  read(Width);
  read(Price);
  writeln('Length= ',Length,'Width= ',Width,'Price= ', Price);
  Cost := Length * Width * Price;
  writeln('Cost:= ', Cost);
END
```

There are plenty of errors here, if you compile the program you’d probably find something like the following list of errors from the compiler.
16 The CONST definition.

Many programs use variables which should never be changed. For example in our area of a circle program earlier we never need to change the value of Pi. To prevent accidentally changing a value that shouldn’t be changed we use a CONST definition. Look at a newer, perhaps more useful of the area of a circle. Unlike the VAR construct you don’t need to specify the type with CONST.

```
PROGRAM CircleArea (input,output)
CONST Pi = 3.14159;
VAR Radius, Area :real;
BEGIN
  (* Get the Radius from the user *)
  writeln('What is the radius of the circle?');
  readln(Radius);
  (* Calculate the Area and display the result *)
  Area := Pi * Radius * Radius;
  writeln('The area of a circle of radius ',Radius, ' =',Area)
END.
```

If we had a line in the program, say,

```
Pi := 6.23;
```

The compiler would give us an error.

When we use the writeln() procedure to print out strings of characters, ie. lines like,

```
writeln('What is the radius of the circle?');
```

the portion in quotes is known as a constant string.

17 Some example programs.

Let’s look at two examples of developing small pascal programs. This should help you see how to design your programs.

Our examples are:

17.1 Computing your GPA

We wish to compute a students GPA. This is done by multiplying the students grade for a course by the number of credits for that course to get a course score. Then we average all the course scores. We assume the student only takes four courses for this problem.

17.1.1 A first pass at the design.

The design goes as follows:

```
PROGRAM GPA;
BEGIN
  Compute the course score for the first course;
  Compute the course score for the second course;
  Compute the course score for the third course;
  Compute the course score for the fourth course;
  Average the course scores;
  Print the averaged scores
END.
```

17.1.2 Refining the design.

The design of each statement of the form

```
Compute the course score for the X course;
```

can be refined to:

```
Read in the course grade;
Read in the number of credits;
Compute the course score;
Assign the value of the Xth score to the computed value;
```

Finally the line

```
Average the course scores
```
becomes

Add the 1st, 2nd, 3rd and 4th course scores and divide them by the total number of credits.

Now we see that we need to have kept track of the total number of credits entered. This is done by adding the line

```
Update the total number of credits;
```

to

```
Read in the course grade;
Read in the number of credits;
Compute the course score;
Assign the value of the Xth score to the computed value;
```

Which leaves us with

```
Read in the course grade;
Read in the number of credits;
Compute the course score;
Update the total number of credits;
Assign the value of the Xth score to the computed value;
```

17.1.3 The pseudocode becomes...

So now our pseudocode looks like

```
PROGRAM GPA;
BEGIN
  (* Com-pute the course score for the first course *)
  Read in the course grade;
  Read in the number of credits;
  Compute the course score;
  Update the total number of credits;
  Assign the value of the Xth score to the computed value;

  (* Com-pute the course score for the second course *)
  Read in the course grade;
  Read in the number of credits;
  Compute the course score;
  Update the total number of credits;
  Assign the value of the Xth score to the computed value;

  (* Com-pute the course score for the third course *)
  Read in the course grade;
  Read in the number of credits;
  Compute the course score;
  Update the total number of credits;
  Assign the value of the Xth score to the computed value;

  (* Com-pute the course score for the fourth course *)
  Read in the course grade;
  Read in the number of credits;
  Compute the course score;
  Update the total number of credits;
  Assign the value of the Xth score to the computed value;

  (* Do the averaging *)
  Add the 1st, 2nd, 3rd and 4th course scores and divide them by the total number of credits.
  Print the result
END.
```

17.1.4 Choosing variables.

What variables do we need?

We need two input variables, one for each course grade and one for the number of credits.

We need four variables for course scores.

We need a variable for the total number of course credits taken.

We need a variable for the gpa.

17.1.5 Our last piece of pseudocode then becomes.

```
PROGRAM GPA;
VAR InputGrade, InputCredits:real; (* input variables *)
Grade1,Grade2,Grade3,Grade4:real;(* four course scores *)
TotalCredits, GrPtAverage:real;
BEGIN
  (* Com-pute the course score for the first course *)
  Read in the course grade;
  Read in the number of credits;
  Compute the course score;
  Update the total number of credits;
  Assign the value of the Xth score to the computed value;

  (* Com-pute the course score for the second course *)
  Read in the course grade;
  Read in the number of credits;
  Compute the course score;
  Update the total number of credits;
  Assign the value of the Xth score to the computed value;

  (* Com-pute the course score for the third course *)
  Read in the course grade;
  Read in the number of credits;
  Compute the course score;
  Update the total number of credits;
  Assign the value of the Xth score to the computed value;

  (* Com-pute the course score for the fourth course *)
  Read in the course grade;
  Read in the number of credits;
  Compute the course score;
  Update the total number of credits;
  Assign the value of the Xth score to the computed value;

  (* Do the averaging *)
  Add the 1st, 2nd, 3rd and 4th course scores and divide them by the total number of credits.
  Print the result
END.
```
Read in the course grade; Read in the number of credits; Compute the course score; Update the total number of credits; Assign the value of the Xth score to the computed value; 

(* Compute the course score for the first course *)

Read in the course grade; Read in the number of credits; Compute the course score; Update the total number of credits; Assign the value of the Xth score to the computed value;

(* Compute the course score for the second course *)

Read in the course grade; Read in the number of credits; Compute the course score; Update the total number of credits; Assign the value of the Xth score to the computed value;

(* Compute the course score for the third course *)

Read in the course grade; Read in the number of credits; Compute the course score; Update the total number of credits; Assign the value of the Xth score to the computed value;

(* Compute the course score for the fourth course *)

Read in the course grade; Read in the number of credits; Compute the course score; Update the total number of credits; Assign the value of the Xth score to the computed value;

(* Do the averaging *)

Add the 1st, 2nd, 3rd and 4th course scores and divide them by the total number of credits.

Print the result

END.

17.1.6 The Pascal program for GPA.

PROGRAM GPA(input,output);
VAR InputGrade, InputCredits:real; (* input variables *)
Grade1,Grade2,Grade3,Grade4:real;(* four scores *)
TotalCredits, GrPtAverage:real;
BEGIN
(* Initialize TotalCredits *)
TotalCredits := 0.0;

(* Compute the course score for the first course *)
Write(’Enter the course grade for the first course ’);
Readln(InputGrade);
Write(’Enter the credit hours for the first course ’);
Readln(InputCredits);
TotalCredits := TotalCredits + InputCredits;
Grade1 := InputGrade * InputCredits;

(* Compute the course score for the second course *)
Write(’Enter the course grade for the second course ’);
Readln(InputGrade);
Write(’Enter the credit hours for the second course ’);
Readln(InputCredits);
TotalCredits := TotalCredits + InputCredits;
Grade2 := InputGrade * InputCredits;

(* Compute the course score for the third course *)
Write(’Enter the course grade for the third course ’);
Readln(InputGrade);
Write(’Enter the credit hours for the third course ’);
Readln(InputCredits);
TotalCredits := TotalCredits + InputCredits;
Grade3 := InputGrade * InputCredits;

(* Compute the course score for the fourth course *)
Write(’Enter the course grade for the fourth course ’);
Readln(InputGrade);
Write(’Enter the credit hours for the fourth course ’);
Readln(InputCredits);
TotalCredits := TotalCredits + InputCredits;
Grade4 := InputGrade * InputCredits;

(* Do the averaging *)
GrPtAverage := (Grade1 + Grade2 + Grade3 + Grade4) / TotalCredits;
WriteLn(’The GPA for 4 courses is ’,GrPtAverage);
END.