The IF-THEN-ELSE and CASE statements.

So far our programs have behaved in fairly linear way. We run every instruction in the main body of the program in sequence from top to bottom being sure to obey the semantics of the looping syntax and the procedure calls encountered.

The problem with this it’s fairly atypical. We don’t really write programs, or solve problems in only one predetermined way everytime we run a program. We need to be able to solve more general problems that have an element of choice in the what statement they execute.

We need a way to control if some statements are to be executed at all. This is done using the IF-THEN-ELSE or CASE syntax of Pascal.

1 A few motivating examples.

When the Pentagon call you up and say “Build us a system that launches a nuclear response.” you can write a program that checks to see “if” there are missiles heading your way. The pseudocode could be something like

```
IF missiles are heading our way THEN
BEGIN
  launch a response;
  head for the hills;
END.
```

Naturally “missiles are heading our way”, “launch a response” and “head for the hills” are well defined procedures you’ve already written using top down design.

You may need to write other programs with many conditions such as a medical diagnostic program that chooses appropriate actions based on the age, sex and previous health record of a person.

2 Conditional statements.

We can do all this using the techniques we learn in this chapter. The focus here is to learn about statements that alter the execution of a program based on the programs input parameters. These statements are called conditional statements. There are two basic forms, the IF-THEN statement and the CASE statement.

3 The IF-THEN statement.

3.1 Boolean Expressions and Comparison Operators.

A boolean expression is an expression whose value returns either TRUE or FALSE.

A trivial boolean expression is TRUE or FALSE.

We can be more expressive than this. We could do tests like

```
A = 30
```

which asks the question, “Is A equal to 30?”. If the answer is yes then the expression evaluates to TRUE.

Boolean expressions are an integral part of chosing a course of action. They form the IF part of the conditionals.

3.2 The basic comparison operators.

These are often used for forming boolean expressions.
Equality
<> Not Equal
< Less Than
> Greater Than
<= Less Than or Equal to.
>= Greater Than or Equal to.

e.g.

2 > 7 == False
2 < 7 == True
2 <= 7 == True
2 <> 7 = True

3.3 Choosing a course of action.

The IF–THEN statement allows us to chose whether to execute a set of program statements based on a test.

The format of the IF–THEN statement is,

```
IF Conditional Expression THEN
  Statement
```
or

```
IF Conditional Expression THEN
  Compound Statement
```

This can be read as follows. IF the Conditional Expression is true THEN do the statement, or compound statements that follows.

3.4 The census problem revisited.

Take the census problem. We want to determine what percentage of the population of a city in a survey are from the north, east, west or south of a city.

Here is a simple version of the program.

```pascal
PROGRAM SimpleSurvey(input,output);
  CONST SampleSize = 100;
  VAR N, E, W, S:integer
    Counter:integer;
    Direction:character;
  BEGIN
    (* Initialize counters *)
    N := 0; E := 0; W := 0; S := 0;
    FOR Counter := 1 TO SampleSize DO
      BEGIN
        write('Please tell me where you come from.
            Type N,E,W or, S followed by [return]');
        readln(Direction);

        (* Increases the correct
        IF Direction = ‘N’ THEN
          N := N + 1;
```
IF Direction = 'E' THEN
    E := E + 1;
IF Direction = 'W' THEN
    W := W + 1;
IF Direction = 'S' THEN
    S := S + 1;
END; (* FOR Counter *)

writeln('Of ', SampleSize, ' people');
writeln(N, ' came from the north');
writeln(E, ' came from the east');
writeln(W, ' came from the West');
writeln(S, ' came from the South');
write('and ', 100 - (N + E + W + S))
writeln(' people answered incorrectly')
END.

### 3.5 Another version of the Census program.

This version of the census program is more aggressively top down in nature. It is here only to serve as an example of how procedures are used in programs.

PROGRAM Survey(input, output);

CONST SampleSize = 100;

VAR North, East, West, South: integer;
    Count: integer;

PROCEDURE AskAndRecord(VAR N, E, W, S: integer);
    VAR WhereFrom: Char;

PROCEDURE Ask(VAR Direction: char);
BEGIN
    write('Please tell me where you come from. Type N, E, W or, S followed by [return]'');
    readln(Direction);
END;

PROCEDURE Record(VAR Direction: Char);
BEGIN
    IF Direction = 'N' THEN
        N := N + 1;
    IF Direction = 'E' THEN
        E := E + 1;
    IF Direction = 'W' THEN
        W := W + 1;
    IF Direction = 'S' THEN
        S := S + 1;
    END;
BEGIN
    Ask(WhereFrom);
    Record(WhereFrom);
END; (* AskAndRecord *)

BEGIN
    North := 0; East := 0; West := 0; South := 0;

    AskAndRecord(North, East, West, South);
BEGIN
writeln('Beginning the survey.')
FOR Count := 1 TO SampleSize DO
    AskAndRecord(North,East,South,West);
writeln('Of ', SampleSize, ' people');
writeln(North, ' came from the north');
writeln(East , ' came from the east');
writeln(West , ' came from the West');
writeln(South, ' came from the South');
write('and ', 100 - (North + East + West + South));
writeln(' people answered incorrectly');
END.

4 An introduction to word processing.

Were going to write some simple programs to illustrate the use of some of the concepts we’ve learned so far. (We use the term word processing above rather loosely!)

The first couple of programs we write will solve the following problems:

- We write a program that counts the number of words in a sentence in which only a blank statement can separate the words.
- We modify the above program to calculate the average length of letters per word in a sentence.

Let’s tackle the first program.

The initial pseudocode is:

Prompt the user to type the # of characters in the sentence.
Label the columns.
Count the number of blanks in the sentence.
Print the results.

The top down diagram looks like:

```
MAIN
   | +-----------------+-----------------+
   | Prompt           Label Cols
   | +-----------------+-----------------+
   | ReadAndCount     Print
```

The pseudocode for the procedure ReadAndCount should look something like,

```
BlankCount := 0;
FOR each character in sentence DO
    IF character = Blank THEN
        BlankCount := BlankCount +1;
```

What we do is count the number of words in a sentence by counting the number of blanks between the words.

e.g.
The_sentence_foofy_has_six_words._
There are six blanks here.
4.1 Prompt

We basically only need to know how long a sentence is now, for that we write the procedure Prompt below.

```pascal
PROCEDURE Prompt(VAR Length:integer);
BEGIN
  writeln('How many columns in the sentence?');
  readln(Length);
  writeln('Type the sentence');
END.
```

4.2 ReadAndCount

We can now write exact code for the ReadAndCount procedure. We read the buffer character by character until we reach the number of characters we specified above, then when we see a blank we increment the blank count. The number of words is the blank count + 1.

```pascal
PROCEDURE ReadAndCount(VAR WordCount:integer; Length:integer);
CONST Blank = ' ';
VAR Character:char;
  BlankCount, J:integer;
BEGIN
  BlankCount := 0;
  FOR J := 1 TO Length DO
    BEGIN
      read (Character);
      IF Character = Blank THEN
        BlankCount := BlankCount +1;
    END; (* For J *)
  WordCount := BlankCount + 1;
END;
```

4.3 LabelColumns

To fill in the rest of the program we write LabelColumns as follows.

```pascal
PROCEDURE LabelColumns(Length:integer);
VAR I, J:integer;
BEGIN
  FOR I := 1 TO Length DO
    BEGIN
      write (I MOD 10:1);
      writeln;
    FOR J := 1 TO Length DO
      BEGIN
        write (J DIV 10:10);
        writeln;
      END.
END.
```

This produces the output

```
000000000111111111122222...
123456789012345678901234...
```
4.4 A better LabelColumns.

Actually, we can do better, suppose we wanted the output to look less cluttered, i.e. look something like:

```
0 1 2 ...
123456789012345678901234...
```

We need only print the variables on the top line ten spaces apart, or everytime the 10’s column changes.

We write the less cluttered version of Label as follows:

```pascal
PROCEDURE LabelColumns(Length:integer);
VAR I, J, LastOut,ThisCount:integer;
BEGIN
  LastOut = -1
  FOR I := 1 TO Length DO
  BEGIN
    ThisCount := I DIV 10;
    (* Checks to see if we’ve
    already printed a number *)
    IF ThisCount <> LastOut THEN
    BEGIN
      LastOut := ThisCount;
      write (LastOut:1);
    END;
    writeln;
  END;
  FOR J := 1 TO Length DO
  write (J MOD 10:10);
  writeln;
END.
```

The trick above is to keep track of the last value printed. If that value changes we print the new value and set the last printed value to the new value.

4.5 ReadAndCount again.

We could use the idea of keeping track of past values to make our ReadAndCount program deal with more than one space between words on our sentences. If your program is more flexible it’s better.

Here we keep track of the last character read. If we read in a blank followed by a blank we can decrement the blank count to prevent us from overcounting.

```pascal
PROCEDURE ReadAndCount(VAR WordCount:integer; Length:integer);
CONST Blank = ' ';
VAR LastChar, Character:char;
  BlankCount, J:integer;
BEGIN
  BlankCount := 0; LastChar := chr(0);
  FOR J := 1 TO Length DO
  BEGIN
    read (Character);
    IF Character = Blank THEN
      BlankCount := BlankCount + 1;
    IF (LastChar = Character) AND
      (LastChar <> Blank ) THEN
```
BlankCount := BlankCount - 1;
LastChar := Char;
END; (* For J *)
WordCount := BlankCount + 1;
END;

4.6 The main program.

So now the main program simply becomes:

PROGRAM Word1(input,output);
VAR Length, WordCount:integer;

PROCEDURE Prompt(VAR Length:integer);
PROCEDURE LabelColumns(Length:integer);
PROCEDURE ReadAndCount(VAR WordCount:integer;
Length:integer);
PROCEDURE Print(WordCount:integer);

BEGIN
(* Get and Set Length *)
Prompt(Length);

(* Draw a rule Length long *)
LabelColumns(Length);

(* Read and Count the sentence *)
ReadAndCount(WordCount,Length);

(* Print *)
Print(WordCount:integer)
END.

All the procedures are as declared above.

5 The IF-THEN-ELSE statement.

We have a statement so far that says IF a condition holds THEN take some action.
Pascal gives us another form of the IF-THEN statement. This allows us to take one of two paths through a set of statements.
The syntax of the IF-THEN-ELSE statement is as follows.

IF Conditional Expression THEN
   Statement1 (or CompoundStatement1)
ELSE
   Statement2 (or CompoundStatement2)

The IF-THEN-ELSE statement says: IF some condition is TRUE THEN execute the block of code following the THEN part, otherwise execute the block of code following the ELSE part.
This differs significantly from the IF-THEN statement. Why?
If we chose to count how many males and females there were in a room we could have a program that said:

PROGRAM GenderCounter(input,output);
CONST SampleSize = 100;
VAR Male, Female, Counter:integer;
    Response:char;
BEGIN

Male :=0; Female :=0;

FOR Counter := 1 TO SampleSize DO
BEGIN
    write('What is your gender? [M/F]');
    readln(Response);
    IF Response = 'F' THEN
        Female := Female +1;
    ELSE
        Male := Male + 1;
    END

Writeln('There are ', Female,
      ', Female, ' females and ', Male,
      ', males in the survey')
END.

6  Calculating the average word length.

We return to our word processor example again. Before we get to calculating the average word length.

Before we do that however let’s get comfortable with this new piece of syntax by modifying the LabelColumns procedure to use the IF-THEN-ELSE statement.

6.1 Modifying LabelColumns again.

This version of the LabelColumns procedure uses the IF-THEN-ELSE statement to print the desired output.

PROCEDURE LabelColumns(Length:integer);
    CONST Blank = ' ';
    VAR I, J:integer;
BEGIN
    FOR I := 1 TO Length DO
    BEGIN
        IF (I MOD 10) = 0 THEN
            write (I);
        ELSE
            write (Blank);
        writeln;
    FOR J := 1 TO Length DO
        write (J MOD 10:10);
    writeln;
    END.

It’s easy to see that this produces,

0   1   2 ...
123456789012345678901234...

It says that “If I MOD 10 is equal to 0 print the value of I otherwise print a blank”.

This is considerably simpler than the version we had before.
6.2 Modifying ReadAndCount to calculate the average word length.

We need to calculate the average word length. Read and count works based around these lines in the loop.

FOR each letter of the sentence DO
  IF Character = Blank THEN
    BlankCount := BlankCount + 1;
  ELSE
    LetterCount := LetterCount + 1;

If we want to keep track of word length we could modify the above as follows:

FOR each letter of the sentence DO
  IF Character = Blank THEN
    BlankCount := BlankCount + 1;
  ELSE
    LetterCount := LetterCount + 1;

Then later on in the code we need only add the following line.

AverageWordLength := LetterCount / WordCount;

to get the answer we wanted.

We need to modify the procedure header to take a third argument so that we can get back enough information to get the average word length.

So now looking at the whole procedure ReadAndCount

PROCEDURE ReadAndCount(VAR WordCount, LetterCount:integer
  ; Length:integer);
CONST Blank = ‘ ’;
VAR LastChar, Character:char;
  BlankCount, J:integer;
BEGIN
  BlankCount := 0; LastChar := chr(0);
  FOR J := 1 TO Length DO
    BEGIN
      read (Character);
      IF Character = Blank THEN
        BlankCount := BlankCount + 1;
      ELSE
        LetterCount := LetterCount + 1;
      END; (* For J *)
      IF LastChar = Character THEN
        BlankCount := BlankCount - 1;
        LastChar := Char;
    END; (* For J *)
  WordCount := BlankCount + 1;
END;

PROCEDURE Print(WordCount, LetterCount:integer);
VAR Average:real;
BEGIN
  writeln(’There are’, WordCount,’ words in the sentence.’);
Average := LetterCount / WordCount;
writeln('The total # of non blanks is ', LetterCount:2);
writeln('The average WordLength is ', Average:6:2);

The main program looks like this:

PROGRAM Wordl(input, output);
VAR Length, WordCount, LetCount:integer;
PROCEDURE Prompt(VAR Length:integer);
PROCEDURE LabelColumns(Length:integer);
PROCEDURE ReadAndCount(VAR WordCount:integer;
Length:integer);
PROCEDURE Print(WordCount:integer);
BEGIN
  (* Get and Set Length *)
  Prompt(Length);
  (* Draw a rule Length long *)
  LabelColumns(Length);
  (* Read and Count the sentence *)
  ReadAndCount(WordCount, LetCount, Length);
  (* Print *)
  Print(WordCount, LetCount:integer)
END.

7 The Boolean operators AND, OR, and NOT.

Often the conditional expressions in an IF-THEN-ELSE statement is more than just a simple comparison operation. Often it can be a set of different conditions.

We can create more complex conditional statements by using the AND, OR, and NOT operators.

These are all boolean operators. AND and OR take two boolean expressions as their arguments and have the basic form \( \text{Arg1} \text{ OP } \text{Arg2} \).

NOT takes the result of a boolean expression and inverts it. i.e.

\[
\begin{align*}
\text{NOT TRUE} & \quad \text{is FALSE} \\
\text{NOT FALSE} & \quad \text{is TRUE}
\end{align*}
\]

7.1 The AND operator.

An AND says if BOTH of the arguments are TRUE then the result of the AND expression is TRUE.

These line of code calculate the total number of letters in the range A - Z there are in a sentence.

\[
\text{FOR Count := 1 TO Length DO} \\
\text{Read(Character);} \\
\text{IF (Character >= 'A') AND (Character <= 'Z')} \\
\text{THEN} \\
\text{LetterCount := LetterCount +1;}
\]
The expression above says if the first condition AND the second condition hold, i.e. Character is in the range A - Z, then increment the count. Both conditions must hold for the statement to be executed.

NOTE: If we have a string of boolean expressions ANDed together if any one expression evaluates to FALSE the whole expression will be FALSE.

### 7.2 The OR operator.

An OR says if EITHER of the arguments are TRUE then the result of the OR expression is TRUE.

These line of code calculate the total number of letters in the range A - Z plus the numbers 0,1,2...9 there are in a sentence.

```pascal
FOR Count := 1 TO Length DO
    Read(Character);
    IF ((Character >= '0') AND (Character <= '9')) OR ((Character >= 'A') AND (Character <= 'Z'))
    THEN
        LetterCount := LetterCount +1;
```

Here we check if the Character is in the range of capital letters OR the range of numbers.

NOTE: If we have a string of boolean expressions ORed together if any one expression evaluates to TRUE the whole expression will be TRUE.

### 7.3 The NOT operator.

A NOT simply flips the value of a boolean expression from TRUE to FALSE.

These line of code calculate the total number of characters which aren’t in the range A - Z.

```pascal
FOR Count := 1 TO Length DO
    Read(Character);
    IF (NOT ((Character >= 'A') AND (Character <= 'Z')))
    THEN
        LetterCount := LetterCount +1;
```

### 7.3.1 Some practice ...

What is the value of the following expressions?

- TRUE AND TRUE
- TRUE AND FALSE
- NOT (TRUE AND FALSE)
- TRUE AND TRUE AND TRUE AND TRUE
- TRUE AND TRUE AND FALSE AND TRUE
- FALSE OR FALSE OR FALSE OR FALSE
- FALSE OR FALSE OR FALSE OR TRUE
- TRUE AND (2 < 3)
- (NOT FALSE) AND TRUE
- FALSE OR FALSE OR FALSE OR (2 < 3)

### 7.4 A puzzle: This is extra credit.

This is a real puzzler for you to work out so you understand how AND, OR, and NOT work. It will also give you some insight into how some of the computers operations work.
DO NOT DO THIS AT THE EXPENSE OF KEEPING UP IN CLASS. If you can program this problem out you’ll really have learned something in this class. Part four is a hard problem! Part’s 1, 2 and 3 are easier.

When we design digital circuits we use devices called GATES. For our purposes there are three kinds of gates, AND, OR, and NOT.

The AND gate looks like this:

```
+-----+ A B AND
|     |
|A    |
|N +---- Output|
|D |
B +--- A AND B
|    +-----+
```

The OR gate is:

```
+-----+ A B OR
|     |
|A    |
|O +---- Output|
|A OR B |
B +--- +-----+
```

and finally NOT looks like:

```
+-----+ NOT A
|     |
|N |
|O +---- Output|
|T |
A +--- NOT A
|    +-----+
```

We represent a one bit binary number, i.e. the values zero or one, using 0 and 1.

To add two one bit binary numbers we need a table that looks like this:

```
A B SUM Carry
-----------
0 0 0 0
0 1 1 0
1 0 1 0
1 1 0 1
```

This says that adding A and B will give us a value for the added bit and a value for the carry over bit. Confused? Think of decimals, 3 + 7 is 10, 0 is the added value and 1 is carried over into the tens column.

If we think about this carefully we can make a circuit by joining the gates above as follows:

[See extra credit diagram on home page.]
Thus we can add two bits together to get the correct values.

Our problem is to model the behaviour of this diagram.

Now what if we add an input C-in, short for CARRY IN. We should get a table like this.

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C-IN</th>
<th>SUM</th>
<th>Carry</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

1. Can you draw a diagram using gates as in the example above to produce a circuit that takes three values A, B and C-IN and outputs the two values SUM and CARRY? This circuit is called a FULL ADDER.

2. The circuit given above is called a HALF ADDER.

We can write a procedure called HalfAdder that gives us the behaviour of the half adder as follows:

```pascal
PROCEDURE HalfAdder(VAR Sum, Carry:boolean; A, B:boolean);
BEGIN
    Carry := A AND B;
    Sum := (A OR B) AND (NOT Carry);
END;
```

Can you write the procedure FullAdder?

3. By using the C-IN value we can add a two two-bit number call them A and B by drawing a diagram that links the FULL ADDERS in the following way.

Add using a half adder add the smallest bit of A and B. Add the second higher bit of A and B using a full adder with C-IN set to the value of CARRY from the half adder.

Draw this diagram. Extend it to add four bit numbers.

Write a program called AddTwoBitNumbers that uses the procedures HalfAdder and FullAdder.

Write a program called AddFourBitNumbers that uses the procedures HalfAdder and FullAdder that add two four bit numbers. (Easy if you can write AddTwoBitNumbers).

For the above. Represent your numbers as boolean values. i.e. Let 1 be TRUE and 0 be FALSE. So if A is a two bit number then give a name for A’s first bit, call it A1, and make it a boolean. Similarly A’s second bit will be called A2. Before you worry about zeros and ones represent your numbers as TRUE and FALSE. So the number three in binary usually 11 is now TRUE TRUE, two is 10 or TRUE FALSE etc.

4. REALLY IMPRESS ME!

Make AddFourBitNumbers work on a string of characters that represent the numbers.

i.e. 0000 or 1111 or 1010.

8  An introduction to some set operations.

In Pascal a set is defined as a collection of ordinal types. The items in the set are known as elements of the set. This isn’t quite the same as the notion of a set in mathematics because we’re restricted to ordinal types.
8.1 Defining sets.

A set is defined using the set constructors \([\text{elements}]\). e.g. we construct the set of vowels as follows:

\[
[\ 'a', 'e', 'i', 'o', 'u']
\]

Here the elements of the set are the lower case vowels and are each separated by a comma. The square brackets are called set formers.

If we know there is a \textit{collating sequence} for values of an ordinal type, like integers then we can declare some sets as follows.

\[
[ 'A' .. 'M' ]
\]

This declaration declares the set of elements \(A - M\). The two dots .. says, “fill in the elements that come between the first and last element of the set”.

The following is not a legal set declaration,

\[
[ 1.23 .. 1.81 ]
\]

8.2 The \textsc{in} operator.

If we can define sets we obviously need a way to determine what its elements are, this is done by using the \textsc{in} operator. \textsc{in} tests for elements in a set.

The syntax of \textsc{in} is:

\[
e \textsc{in} \text{set}
\]

\(e\) must be a single value and \text{set} must be a set of elements. \textsc{in} yield a binary value, \textsc{true} or \textsc{false} depending on whether \(e\) is an element of \text{set} or not.

Some examples are in order:

\[
3 \textsc{in} [1 .. 7] \Rightarrow \textsc{true}
\]

\[
9 \textsc{in} [1 .. 7] \Rightarrow \textsc{false}
\]

\[
'a' \textsc{in} [ 'a', 'e', 'i', 'o', 'u' ] \Rightarrow \textsc{true}
\]

\[
'c' \textsc{in} [ 'a', 'e', 'i', 'o', 'u' ] \Rightarrow \textsc{false}
\]

Because expressions involving \textsc{in} are boolean, i.e. they evaluate to \textsc{true} or \textsc{false} we can use them in \textsc{if-then-else} statement. For example:

\[
\text{FOR Digit := 0 TO 9 DO}
\]

\[
\text{ IF Digit \textsc{in} [ 5 .. 8 ] THEN}
\]

\[
\text{ write(Digit:2);}
\]

This will print the digits 5 6 7 8.

\[
\text{FOR Letter := 'A' TO 'Z' DO}
\]

\[
\text{ IF NOT (Letter \textsc{in} [ 'A', 'E', 'I', 'O', 'U' ] THEN}
\]

\[
\text{ write(Letter);}
\]

This will print the capital consonants.
9 Our Word Processing revisited.

In this part of our wordprocessor we make our wordprocessor calculate average word lengths for sentences with punctuation. This should be easy enough to do using our new set operations.

This was the old version.

```plaintext
FOR J := 1 TO Length DO
BEGIN
read (Character);
IF Character = Blank THEN
    BlankCount := BlankCount + 1;
ELSE
    LetterCount := LetterCount + 1;
IF (LastChar = Character) AND
    (LastChar <> Blank) THEN
    BlankCount := BlankCount - 1;
LastChar := Char;
END; (* For J *)
WordCount := BlankCount + 1;
END;
```

We can turn this into a somewhat neater version:

```plaintext
FOR J := 1 TO Length DO
BEGIN
read (Character);
IF (Character IN ['a' .. 'z']) OR (Character IN ['A' .. 'Z']) THEN
    LetterCount := LetterCount + 1;
IF (LastChar = Character) AND
    (LastChar <> Blank) THEN
    BlankCount := BlankCount + 1;
LastChar := Char;
END; (* For J *)
WordCount := BlankCount + 1;
END;
```

We use \texttt{IN} to increment \texttt{LetterCount} only when we see characters between \texttt{`A’} - \texttt{‘Z’} or \texttt{‘a’} - \texttt{‘z’}.

10 Adding two numbers containing a \$ and a comma: Top down testing.

Often in reading though books, tax forms and your NYU tuition bills, or if you’re lucky enough on your IRS refund, you’ll come across dollar amounts represented in the following way, \$12,345.

We may want a computer to be able to operate on numbers represented like this. To do this we need to be able to read in a number like \$12,345 and treat it as the integer 12345.

What we need to do basically is to convert the dollar representation, into an integer representation. We can do this by reading in the characters in the dollar representation and then ignoring the commas and \$ signs.

Let’s propose we do this by creating a procedure called \texttt{Convert} that reads in a dollar value from the sequence and then returns, to the calling program, the appropriate integer value.

You should be asking yourself “HOW DO I RETURN A VALUE TO THE CALLING PROGRAM?”. You should also know the answer is, “BY USING A VARIABLE PARAMETER”.

\texttt{Convert} is simple to write. In terms of pseudocode it’s:

\texttt{Read in the $ value from the keyboard.}
Read in the characters one by one, ignoring commas and $.
Convert the characters read in to their appropriate integer value.

We assume, for now, that we only deal with amounts less than $99,000. Which allows us to say that convert will only work on strings of up to 7 characters long.

Convert looks like this

```pascal
PROCEDURE Convert(VAR Number:integer)
CONST Quote = ' ' '
VAR Ch:Char;
J:Integer;
BEGIN
    Number := 0;
    writeln('Type in a positive dollar value less than $100,000');
    FOR J := 1 TO 7 DO BEGIN
        read(Ch);
        IF Ch IN ['0' .. '9'] THEN
            Number := Number * 10 + ord(Ch) - ord('0')
        ELSE
            Write(Quote,Ch,Quote, ' encountered: ignoring it!');
    END;
    writeln;
    writeln('Converted number =', Number);
    Readln
END;
```

Basically we loop seven times. If we’re reading in a character in the range ‘0’ – ‘9’ and convert it to the integer in the range 0 – 9. For each time we read in a number multiply the number read in so far by 10 and add in the current number value.

So if we read in the number $72,340 the calculation would go as follows:

<table>
<thead>
<tr>
<th>Loop</th>
<th>Ch</th>
<th>Ord(Ch) - Ord(0)</th>
<th>Number</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>$</td>
<td>ignored</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>7</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>2</td>
<td>72</td>
</tr>
<tr>
<td>4</td>
<td>,</td>
<td>ignored</td>
<td>72</td>
</tr>
<tr>
<td>5</td>
<td>3</td>
<td>3</td>
<td>723</td>
</tr>
<tr>
<td>6</td>
<td>4</td>
<td>4</td>
<td>7234</td>
</tr>
<tr>
<td>7</td>
<td>0</td>
<td>0</td>
<td>72340</td>
</tr>
</tbody>
</table>

There is a subtle problem with this procedure, can you tell what it is?

11 Nested IF statements.

If you remembered that the syntax of the IF-THEN-ELSE statement was

```pascal
IF Conditional Expression THEN
    Statement1 (or CompoundStatement1)
ELSE
    Statement2 (or CompoundStatement2)
```

you will immediately see that the following is perfectly legal piece of Pascal.
IF A > B THEN
  IF C > D THEN
    Write('first');
  ELSE
    Write('second');

The test C > D is only executed if A > B.

The ELSE part always matches with the closes IF that hasn’t been matched. This is important to remember. To get the ELSE above to match the A > B condition, we’d need to use BEGIN and END.

IF A > B THEN
  BEGIN
    IF C > D THEN
      Write('first');
  END
ELSE
  Write('second');

This effectively says that the nested IF statement is in a compound statement.

12 The multiple alternative IF-THEN-ELSE.

Sometimes a test condition may have multiple alternatives. For example we want to work out a persons income tax. The tax may is decided on which tax bracket the person is in. We have a different alternative for each bracket.

The following code segment introduces the idea of multiple alternatives.

income:longint
Convert(income)
IF (income <= 2300) THEN
  Tax := 0;
ELSE IF (income <= 3400) THEN
  Tax := 0.14 * income
ELSE IF (income <= 4400) THEN
  Tax := 0.16 * income
ELSE IF (income <= 6500) THEN
  Tax := 0.18 * income
ELSE IF (income <= 8500) THEN
  Tax := 0.19 * income
ELSE IF (income <= 10800) THEN
  Tax := 0.21 * income
ELSE writeln('Go hire an accountant!');

Why does this work? Here each ELSE matches the most closely unmatched IF. Let’s say income is 4000. The first comparison, i.e. (income <= 2300) in

IF (income <= 2300) THEN

evaluates to FALSE. Therefore we take the ELSE branch, which is an IF-THEN-ELSE statement too.

ELSE
  IF (income <= 3400) THEN
    ...
ELSE
whose test, \((\text{income} \leq 3400)\) also fails. Now because we’ve failed the test above we know that \text{income} is value greater than 3400.

Eventually we pass a test. In this case it’s \((\text{income} \leq 4400)\). Now we take the THEN branch and execute the line inside it.

\[
\text{ELSE IF (income} \leq 4400)\text{ THEN}
\]
\[
\text{Tax} := 0.16 \times \text{income}
\]

What happens with the rest of the alternatives? Do we ever do these tests, after all \((\text{income} \leq 6500)\) evaluates to true as well. Should we be executing the code for that conditional too? Clearly the answer is NO. But we’re OK because we never actually DO the next test because it’s part of an ELSE branch that’s never taken.

To make this painfully clear the above block of code is the same as:

\[
\text{IF (income} \leq 2300)\text{ THEN}
\]
\[
\text{Tax} := 0;
\]
\[
\text{ELSE}
\]
\[
\text{IF (income} \leq 3400)\text{ THEN}
\]
\[
\text{Tax} := 0.14 \times \text{income};
\]
\[
\text{ELSE}
\]
\[
\text{IF (income} \leq 4400)\text{ THEN}
\]
\[
\text{Tax} := 0.16 \times \text{income};
\]
\[
\text{ELSE}
\]
\[
\text{IF (income} \leq 6500)\text{ THEN}
\]
\[
\text{Tax} := 0.18 \times \text{income};
\]
\[
\text{ELSE}
\]
\[
\cdot
\]
\[
\cdot
\]
\[
\cdot
\]
\[
\END.
\]

Finally we end the process by taking the last, i.e. most nested, ELSE which if all the tests fail will be a default value.

If you don’t want a default value then don’t write the last ELSE statement. That way we have a test for every set of actions.

### 13 The CASE statement.

The CASE statement is a more restricted form of a multiple alternative statement. It tends to be a lot less cluttered than the IF-THEN-ELSE statement although it is less flexible.

CASE statements test for matches on ordinal values.

The syntax of the case statement is as follows:

\[
\text{CASE } X \text{ OF}
\]
\[
\text{V1: statement \(\text{or Compound Statement}\);}
\]
\[
\text{V2: statement \(\text{or Compound Statement}\);}
\]
\[
\cdot
\]
\[
\cdot
\]
\[
\cdot
\]
\[
\END.
\]

Where \text{V1} is a set of values.

\text{e.g.}

\[
\text{FOR } X := 1 \text{ TO 4 DO}
\]
\[
\text{CASE } X \text{ OF}
\]
In the code above when \( X = 1 \) the first statement gets selected.

This piece of code checks to see if \( X \) is a value from 1 to 5.

### 14 CASE statements that cover ranges of values.

The CASE statement is different from the IF-THEN-ELSE statement in that only individual ordinal values can be matched. You can’t select statements on reals, nor can use comparison operators like \( \leq \).

```plaintext
read(grade);
IF 0 <= grade < 4 THEN
    writeln('less than four');
ELSE
    writeln('Out of range.');
END;
```

The equivalent CASE statement looks like this.

```plaintext
read(grade);
CASE grade OF
    0,1,2,3:writeln('less than four');
ELSE
    writeln('four or more');
END;
```

### 15 An elementary calculator.

Using conditional statements we’ll now build a program to emulate a simple calculator.

Our calculator will read in mathematical expressions and return the answer to the screen. E.g.

```
2 * 3
= 6
```

The operators our calculator will recognise are *, +, -, /.

The numbers our calculator will recognise are integers.

#### 15.1 The pseudocode.

The pseudocode for our calculator looks something like:

```plaintext
Readln(Operand1, operator, Operand2);
Convert Operand1 to a digit;
Convert Operand2 to a digit;
CASE operator OF
    plus: do addition;
```
minus: do subtraction;
mult: do multiplication;
div: do division;
ELSE
    write an error message;
END.
write results;

15.2 Converting the digits to Characters.

We all know by now how to convert a digit to a character.

PROCEDURE Convert(Oper:char;VAR Digit:integer);
BEGIN
    Digit := ord(Oper) - ord('0');
END;

15.3 The calculator.

Now we write the calculator program, from the pseudocode it’s trivial!

PROGRAM Calculator(input,output);
VAR Operand1, Operand2, Operator:char;
    Result, Digit1, Digit2:integer;
PROCEDURE Convert(Oper:char;VAR Digit:integer);
BEGIN
    Digit := ord(Oper) - ord('0');
END;
BEGIN
    Writeln('Type your three character expression!');
    readln(Operand1,Operator,Operand2);
    Convert(Operand1,Digit1);
    Convert(Operand2,Digit2);
    CASE Operator OF
        '+' : Result := Digit1 + Digit2;
        '-' : Result := Digit1 - Digit2;
        '*' : Result := Digit1 * Digit2;
        '/' : Result := Digit1 DIV Digit2;
    ELSE
        writeln('You entered an illegal operator!');
    END (*CASE*)
    writeln('= ', Result);
END

15.4 Adding a power procedure.

Writing a power procedure is easy. We can add it to our calculator with ease too. Here is the code for the Power procedure.

PROCEDURE Power(VAR Result:integer; Base , Power:integer);
    Temp, Counter:integer;
BEGIN
    Temp := 1;
    FOR Counter = 1 TO Power DO
To integrate it into our calculator we just need to choose a new symbol to represent the Power function. Let’s use `^`.

```
CASE Operator OF
  '+' : Result := Digit1 + Digit2;
  '-' : Result := Digit1 - Digit2;
  '*' : Result := Digit1 * Digit2;
  '/' : Result := Digit1 DIV Digit2;
  '^' : Power(Result, Digit1, Digit2);
ELSE
  writeln('You entered an illegal operator!');
END;
```

So now `2^3` will give the result `8`.

### 16 Functions

The other type of subprogram available to you as Pascal programmers are **FUNCTIONS**.

We’ve already used functions. Subprograms like `ord()` that can be used in an assignment statement are basically functions. e.g. `A := ord('p')`.

The difference between procedures and functions is that functions return a value. In the example above the value of the ordinal of `‘p’` appears on the left hand side of the assignment expression.

Functions in Pascal are very close to their mathematical counterparts. In fact, it would have been more natural for us to have written:

```
^ : result := Power(digit1, digit2);
```

instead of

```
^ : Power(result, digit1, digit2);
```

To convert Power from a procedure to a function is easy. This is the function `Power`.

```
FUNCTION Power(Base, Power:integer):integer;
  Temp, Counter:integer;
BEGIN
  Temp := 1;
  FOR Counter = 1 TO Power DO
    Temp := Temp * Base;
  Power := Temp
END;
```

There are some important differences here.

- The keyword **FUNCTION** replaces the keyword **PROCEDURE**.
- We add to the **FUNCTION** header the type of the value that the function returns. This is important if our language is going to continue to be type safe.
- Lastly, the final value returned by the function is assigned to the function name. If we want the function `Power` to return a value we need to assign a value to `Power` in the subprogram.
The nice thing about functions is that now information can flow between the main program and the subprogram but we don’t need to \texttt{VAR} parameters.

If you make sure you use only local variables in your function and pass in no \texttt{VAR} parameters you can minimize any possibility of side-effects occurring in your programs. It’s for this reason that we prefer, for the most part, functions over procedures.

17 Some fun with functions.

Let’s look at a program to calculate the factorial of a number.

The factorial of a number \(N\) is given by the following formula:

\[
\text{Factorial}(N) = N \times (N - 1) \times (N - 2) \times (N - 3) \times \ldots \times 3 \times 2 \times 1
\]

with the special case

\[
\text{Factorial}(0) = 1
\]

If we had to write a program \texttt{Factorial} we could do it by simple iteration.

\begin{verbatim}
FUNCTION Factorial(N:integer):integer;
  VAR Result:integer;
  BEGIN
    FOR Count := 1 TO N DO
      Result := Result * Count;
    Factorial := Result
  END.
\end{verbatim}

Which is pretty easy to do. But here's a more powerful idea. If we look the definition of factorial more closely we see that

\[
\text{Factorial}(N) = N \times (N - 1) \times (N - 2) \times (N - 3) \times \ldots \times 3 \times 2 \times 1
\]

is really the same thing as

\[
\text{Factorial}(N) = N \times \text{Factorial}(N -1);
\]

after all \(\text{Factorial}(N-1)\) is given by

\[
\text{Factorial}(N-1) = (N - 1) \times (N - 2) \times (N - 3) \times \ldots \times 3 \times 2 \times 1
\]

We can imitate this in Pascal easily.

\begin{verbatim}
FUNCTION Factorial(N:integer):integer;
  BEGIN
    IF N = 0 THEN
      Factorial := 1;
    ELSE
      Factorial := N * Factorial(N -1);
    END
\end{verbatim}

We’ve written a Pascal subprogram that uses itself to produce the answer. This powerful mechanism is called \textit{recursion}.

Here's another example of recursion. We want to test if a number is even or odd.

The code for this uses two programs that call each other to produce the result.
FUNCTION Odd(N):boolean;
FUNCTION Even(N:integer):boolean;
BEGIN
  IF N = 1 Then
    Odd := True;
  ELSE IF N = 0 THEN
    Odd := False;
  ELSE
    Odd := Even(N-1);
  END.

FUNCTION Even(N:integer):boolean;
FUNCTION Odd(N:integer):boolean;
BEGIN
  IF N = 1 Then
    Even := False;
  ELSE IF N = 0 THEN
    Even := TRUE
  ELSE
    Even := Odd(N-1);
  END.

Trace this through for yourself and see that it works properly.