Datatypes

1. INTEGER: (0-9) without decimal e.g. 95, 94, 93, 21
2. REAL:(0-9) with decimal e.g. 95.1, 10.0, 9.3, 1.5
3. STRING: Alphabets, Numbers, symbols “This is Computer”
4. CHAR: single character ‘A’, ‘m’, ‘5’
5. BOOLEAN: Comparison True/False

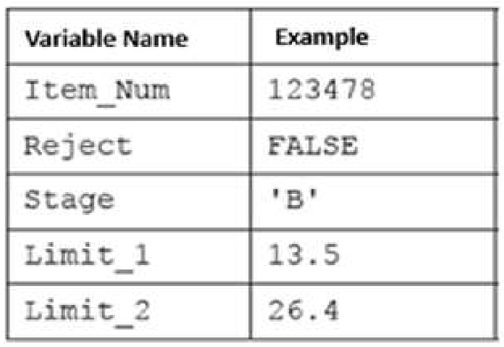
Variable declarations:

DECLARE <identifier> : <data type>

Example

DECLARE Counter : INTEGER
DECLARE TotalToPay : REAL
DECLARE GameOver : BOOLEAN
DECLARE n1, n2, n3 : INTEGER

Declare variables with correct datatype.



Variable Name	Example
Item_Num	123478
Reject	FALSE
Stage	'B'
Limit_1	13.5
Limit_2	26.4

DECLARE Item Num: INTEGER

Constants:

CONSTANT <identifier> = <value>
Example
CONSTANT HourlyRate = 6.50
CONSTANT DefaultText = "N/A"

Assignments:

The assignment operator is ←

Example
Counter ← 0
Counter ← Counter + 1
TotalToPay ← NumberOfHours * HourlyRate

Assigning a value

```
Number ← 1
```

Updating a value

```
Number ← Number + 1
```

Copying a value

```
Number1 ← Number2
```

Swapping two values

```
Temp ← Value1
Value1 ← Value2
Value2 ← Temp
```

Output Information:

```
OUTPUT <string>
```

Example:

```
OUTPUT "Hello World"
```

Concatenate and new line

```
OUTPUT "Hello!", YourName, "Your number is ", Number   // new line
OUTPUT "Hello"          // no new line
```

Getting Input from User:

```
INPUT <identifier>
```

Example:

```
INPUT "Enter miles:" Miles
Km <- Miles * 1.61
OUTPUT "km:", Km
```

Q. There are three sides of triangle a, b and c. Prompt and input sides and show output as given below for the next two sides.

```
OUTPUT "Input length of the first side"
INPUT a
```

Q. Declare suitable variables and write a code for perimeter of triangle.

Final output should be Perimeter is : 1234

```
DECLARE a, b, c, perimeter: INTEGER

a←10

b←20

c←30

perimeter <-- a+b+c

OUTPUT "Perimeter is :", perimeter
```

Q. Declare suitable variables and write a code for perimeter of triangle by getting inputs of sides from user. Final output should be Perimeter is : 1234

```
DECLARE a, b, c, perimeter: INTEGER
INPUT a
INPUT b
INPUT c
perimeter ← a+b+c
OUTPUT "Perimeter is:", perimeter
```

Q. Get three numbers from user and find average.

Final output should be Average is : 25.6

```
DECLARE a, b, c : INTEGER
DECLARE average : REAL
INPUT a
INPUT b
INPUT c
average ← (a+b+c)/3
OUTPUT "Average is :", average
```

Operator	Comparison
=	Is equal to
<	Is less than
>	Is greater than
<=	Is less than or equal to
>=	Is greater than or equal to
<>	Is not equal to

Selection: under certain conditions some steps are performed, otherwise different (or no) steps are performed.

IF...THEN statements

In pseudocode the IF...THEN construct is written as:

```
IF <Boolean expression>
  THEN
    <statement(s)>
ENDIF
```

Pseudocode example:

```
IF x < 0
  THEN
    OUTPUT "Negative"
ENDIF
```

IF...THEN...ELSE statements

In pseudocode, the IF...THEN...ELSE construct is written as:

```
IF <Boolean expression>
  THEN
    <statement(s)>
  ELSE
    <statement(s)>
ENDIF
```

Pseudocode example:

```
IF x < 0
  THEN
    OUTPUT "Negative"
  ELSE
    OUTPUT "Positive"
ENDIF
```

Nested IF statements

In pseudocode, the nested IF statement is written as:

```
IF <Boolean expression>
  THEN
    <statement(s)>
  ELSE
    IF <Boolean expression>
      THEN
        <statement(s)>
      ELSE
        <statement(s)>
    ENDIF
ENDIF
```

Pseudocode example:

```
IF x < 0
  THEN
    OUTPUT "Negative"
  ELSE
    IF x = 0
      THEN
        OUTPUT "Zero"
      ELSE
        OUTPUT "Positive"
    ENDIF
ENDIF
```

Declare suitable variable to show person is less than 60 years or not. Get input from user.
Final output should be Your age is less than 60

```
DECLARE age : INTEGER
INPUT age
IF age<60
    THEN
        OUTPUT "Your age is less than 60"
    ELSE
        OUTPUT "Your age is greater than 60"
ENDIF
```

Get input from user data like name and marks obtained and show if marks obtained are less than 40 then fail otherwise pass.
Final output should be You are pass Ali

```
DECLARE name : STRING
DECLARE marks : INTEGER

INPUT name
INPUT marks

IF marks<40
    THEN
        OUTPUT "You are fail, ", name
    ELSE
        OUTPUT "You are pass, ", name
ENDIF
```

Write a code to check number is even or odd.
Final output should be Number is Even

```
DECLARE n : INTEGER
INPUT n
IF n%2 == 0
    THEN
        OUTPUT "Number is Even"
    ELSE
        OUTPUT "Number is Odd"
ENDIF
```

A person is classed as a child if they are under 13 and as an adult if they are over 19. If they are between 13 and 19 inclusive they are classed as teenagers. We can write these statements as logic statements.

- If Age < 13 then person is a child.
- If Age > 19 then person is an adult.
- If Age >= 13 AND Age <= 19 then person is a teenager.

A number-guessing game follows different steps depending on certain conditions. Here is a description of the algorithm.

- The player inputs a number to guess the secret number stored.
- If the guess was correct, output a congratulations message.
- If the number input was larger than the secret number, output message "secret number is smaller".
- If the number input was smaller than the secret number, output message "secret number is greater".

We can re-write the number-guessing game steps as an algorithm in pseudocode:

```
SET value for secret number
INPUT Guess
IF Guess = SecretNumber
   THEN
      OUTPUT "Well done. You have guessed the secret number"
   ELSE
      IF Guess > SecretNumber
        THEN
          OUTPUT "secret number is smaller"
        ELSE
          OUTPUT "secret number is greater"
      ENDIF
ENDIF
```

More complex conditions can be formed by using the logical operators AND, OR and NOT. For example, the number-guessing game might allow the player multiple guesses; if the player has not guessed the secret number after 10 guesses, a different message is output.

```
IF Guess = SecretNumber
  THEN
    OUTPUT "Well done. You have guessed the secret number"
  ELSE
    IF Guess <> SecretNumber AND NumberofGuesses = 10      complex condition
      THEN
        OUTPUT "You still have not guessed the secret number"
      ELSE
        IF Guess > SecretNumber
          THEN
            OUTPUT "secret number is smaller"
          ELSE
            OUTPUT "secret number is greater"
        ENDIF
    ENDIF
ENDIF
```

WORKED EXAMPLE 12.02

Using selection constructs

The problem to be solved: Take three numbers as input and output the largest number.

There are several different methods (algorithms) to solve this problem. Here is one method.

1 Input all three numbers at the beginning.
2 Store each of the input values in a separate variable (the identifiers are shown in Table 12.04).
3 Compare the first number with the second number and then compare the bigger one of these with the third number.
4 The bigger number of this second comparison is output.

See Worked Example 12.03 for another solution.

Identifier	Explanation
Number1	The first number to be input
Number2	The second number to be input
Number3	The third number to be input

Table 12.04 Identifier table for biggest number problem

The algorithm can be expressed in the following pseudocode:

```
INPUT Number1
INPUT Number2
INPUT Number3
IF Number1 > Number2
  THEN
  // Number1 is bigger
    IF Number1 > Number3
      THEN
        OUTPUT Number1
      ELSE
        OUTPUT Number3
    ENDIF
  ELSE
  // Number2 is bigger
    IF Number2 > Number3
      THEN
        OUTPUT Number2
      ELSE
        OUTPUT Number3
    ENDIF
ENDIF
```

WORKED EXAMPLE 12.03

Using selection constructs (alternative method)

The problem to be solved: Take three numbers as input and output the largest number.

This is an alternative method to Worked Example 12.02.

1 Input the first number and store it in `BiggestSoFar`
2 Input the second number and compare it with the value in `BiggestSoFar`.
3 If the second number is bigger, assign its value to `BiggestSoFar`
4 Input the third number and compare it with the value in `BiggestSoFar`
5 If the third number is bigger, assign its value to `BiggestSoFar`
6 The value stored in `BiggestSoFar` is output.

The identifiers required for this solution are shown in Table 12.05.

Identifier	Explanation
`BiggestSoFar`	Stores the biggest number input so far
`NextNumber`	The next number to be input

Table 12.05 Identifier table for the alternative solution to the biggest number problem

The algorithm can be expressed in the following pseudocode:

```
INPUT BiggestSoFar
INPUT NextNumber
IF NextNumber > BiggestSoFar
   THEN
      BiggestSoFar ← NextNumber
ENDIF
INPUT NextNumber
IF NextNumber > BiggestSoFar
   THEN
      BiggestSoFar ← NextNumber
ENDIF
OUTPUT BiggestSoFar
```

Note that when we input the third number in this method the second number gets overwritten as it is no longer needed.

CASE statements

An alternative selection construct is the CASE statement. Each considered CASE condition can be:

- a single value
- single values separated by commas
- a range.

In pseudocode, the CASE statement is written as:

```
CASE OF <expression>
  <value1>            : <statement(s)>
  <value2>,<value3>   : <statement(s)>
  <value4> TO <value5> : <statement(s)>
  .
  .
  OTHERWISE <statement(s)>
ENDCASE
```

The value of `<expression>` determines which statements are executed. There can be as many separate cases as required. The `OTHERWISE` clause is optional and useful for error trapping.

In pseudocode, an example CASE statement is:

```
CASE OF Grade
  "A"      : OUTPUT "Top grade"
  "F", "U" : OUTPUT "Fail"
  "B".."E" : OUTPUT "Pass"
OTHERWISE OUTPUT "Invalid grade"
ENDCASE
```

Repetition: a sequence of steps is performed a number of times. This is also known as iteration or looping.

Count-controlled (FOR) loops

In pseudocode, a count-controlled loop is written as:

```
FOR <control variable> ← s TO e STEP i // STEP is optional
    <statement(s)>
NEXT <control variable>
```

The control variable starts with value s, increments by value i each time round the loop and finishes when the control variable reaches the value e.

```
FOR x ← 1 TO 5
    OUTPUT x
NEXT x
```

Output

```
1
2
3
4
5
```

```
FOR x ← 1 TO 10 STEP2
    OUTPUT x
NEXT x
```

Output

```
1
3
5
7
9
```

```
FOR x ← 5 TO 1 STEP -1
    OUTPUT x
NEXT x
```

Output

```
5
4
3
2
1
```

Calculate sum of first five numbers.

```
sum ← 0
FOR i ← 1 TO 5
   sum ← sum + i
NEXT i
PRINT "The sum is:", sum
```

Output:

```
The sum is: 15
```

Calculate sum of even numbers.

```
DECLARE sum, i: INTEGER
sum←0
FOR i ← 1 TO 10
    IF i%2==0
        THEN
          sum←sum+i
    ENDIF
NEXT i
OUTPUT "The sum of even numbers are", sum
```

Calculate sum of even and odd numbers.

```
DECLARE EvenNum, OddNum, i: INTEGER
EvenNum ← 0
OddNum ← 0
sum ← 0

FOR i ← 1 TO 10
    IF i%2==0
        THEN
          EvenNum ← EvenNum +i
    ELSE
          OddNum ← OddNum + i
    ENDIF
NEXT i
OUTPUT "The sum of even numbers are", EvenNum
OUTPUT "The sum of odd numbers are", OddNum
```

WORKED EXAMPLE 12.05

Repetition using FOR...NEXT

The problem to be solved: Take 10 numbers as input and output the largest number.

We can use the same identifiers as in Worked Example 12.04. Note that the purpose of `Counter` has changed.

Identifier	Explanation
`BiggestSoFar`	Stores the biggest number input so far
`NextNumber`	The next number to be input
`Counter`	Counts the number of times round the loop

Table 12.07 Identifier table for biggest number problem using a `FOR` loop

The algorithm can be expressed in the following pseudocode:

```
INPUT BiggestSoFar
FOR Counter ← 2 TO 10
      INPUT NextNumber
      IF NextNumber > BiggestSoFar
        THEN
          BiggestSoFar ← NextNumber
      ENDIF
NEXT Counter
OUTPUT BiggestSoFar
```

The first time round the loop, Counter is set to 2. The next time round the loop, Counter has automatically increased to 3, and so on. The last time round the loop, Counter has the value 10.

WORKED EXAMPLE 12.08

Calculating running totals and averages

The problem to be solved: Take 10 numbers as input and output the sum of these numbers and the average.

Identifier	Explanation
`RunningTotal`	Stores the sum of the numbers input so far
`Counter`	How many numbers have been input
`NextNumber`	The next number input
`Average`	The average of the numbers input

Table 12.10 Identifier table for running total and average algorithm

The following pseudocode gives a possible algorithm:

```
RunningTotal ← 0
FOR Counter ← 1 TO 10
      INPUT NextNumber
      RunningTotal ← RunningTotal + NextNumber
NEXT Counter
OUTPUT RunningTotal
Average ← RunningTotal / 10
OUTPUT Average
```

It is very important that the value stored in `RunningTotal` is initialised to zero before we start adding the numbers being input.

Post-condition loops

A post-condition loop, as the name suggests, executes the statements within the loop at least once. When the condition is encountered, it is evaluated. As long as the condition evaluates to False, the statements within the loop are executed again. When the condition evaluates to True, execution will go to the next statement after the loop.

When coding a post-condition loop, you must ensure that there is a statement within the loop that will at some point change the end condition to True. Otherwise the loop will execute forever.

In pseudocode, the post-condition loop is written as:

```
REPEAT
    <statement(s)>
UNTIL <condition>
```

Pseudocode example:

```
REPEAT
    INPUT "Enter Y or N: " Answer
UNTIL Answer = "Y"
```

Calculate sum of 5 numbers.

```
DECLARE sum, i : INTEGER
sum ← 0
i ← 1
REPEAT
        sum ← sum + i
        i ← i + 1
UNTIL i > 5
OUTPUT "The sum of 5 numbers is ", sum
```

An algorithm will: 9618_w23_qp_22

1. Input a sequence of integer values, one at a time

2. Ignore all values until the value 27 is input, then sum the remaining values in the sequence

3. Stop summing values when the value 0 is input and then output the sum of the values.

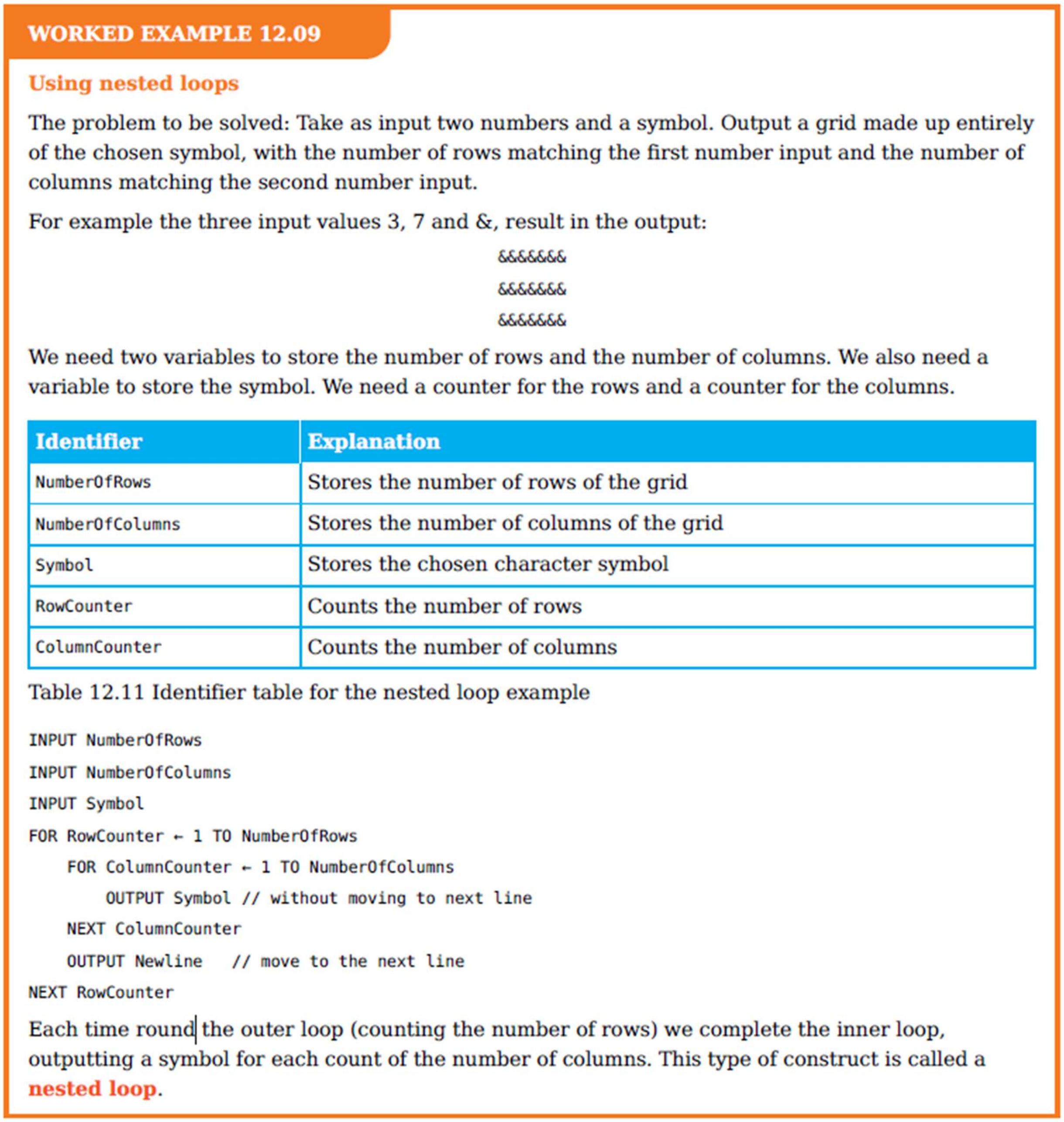
```
DECLARE num, sum : INTEGER
sum ← 0

REPEAT
   INPUT num
UNTIL num = 27

REPEAT
   INPUT num
   IF num <> 0 THEN
      sum ← sum + num
   ENDIF
UNTIL num = 0
OUTPUT "The sum of the values is ", sum
```

1 Name: (pre / post) conditional loop
2 Justification: the number of iterations is not known // loop ends following a specific input (in the loop)

Write pseudocode for the following problem given in structured English.
REPEAT the following UNTIL the number input is zero
INPUT a number
Check whether number is positive or negative
Increment positive number count if the number is positive
Increment negative number count if the number is negative

```
DECLARE number : INTEGER
DECLARE positive_count : INTEGER
DECLARE negative_count : INTEGER

positive_count ← 0
negative_count ← 0

REPEAT
  INPUT number

  IF number > 0 THEN
    positive_count = positive_count + 1
  ELSE
    negative_count = negative_count + 1
  ENDIF

UNTIL number == 0

OUTPUT "Total positive numbers entered: ", positive_count
OUTPUT "Total negative numbers entered: ", negative_count
```

Nested Loops



WORKED EXAMPLE 12.09

Using nested loops

The problem to be solved: Take as input two numbers and a symbol. Output a grid made up entirely of the chosen symbol, with the number of rows matching the first number input and the number of columns matching the second number input.

For example the three input values 3, 7 and &, result in the output:

```
&&&&&&&
&&&&&&&
&&&&&&&
```

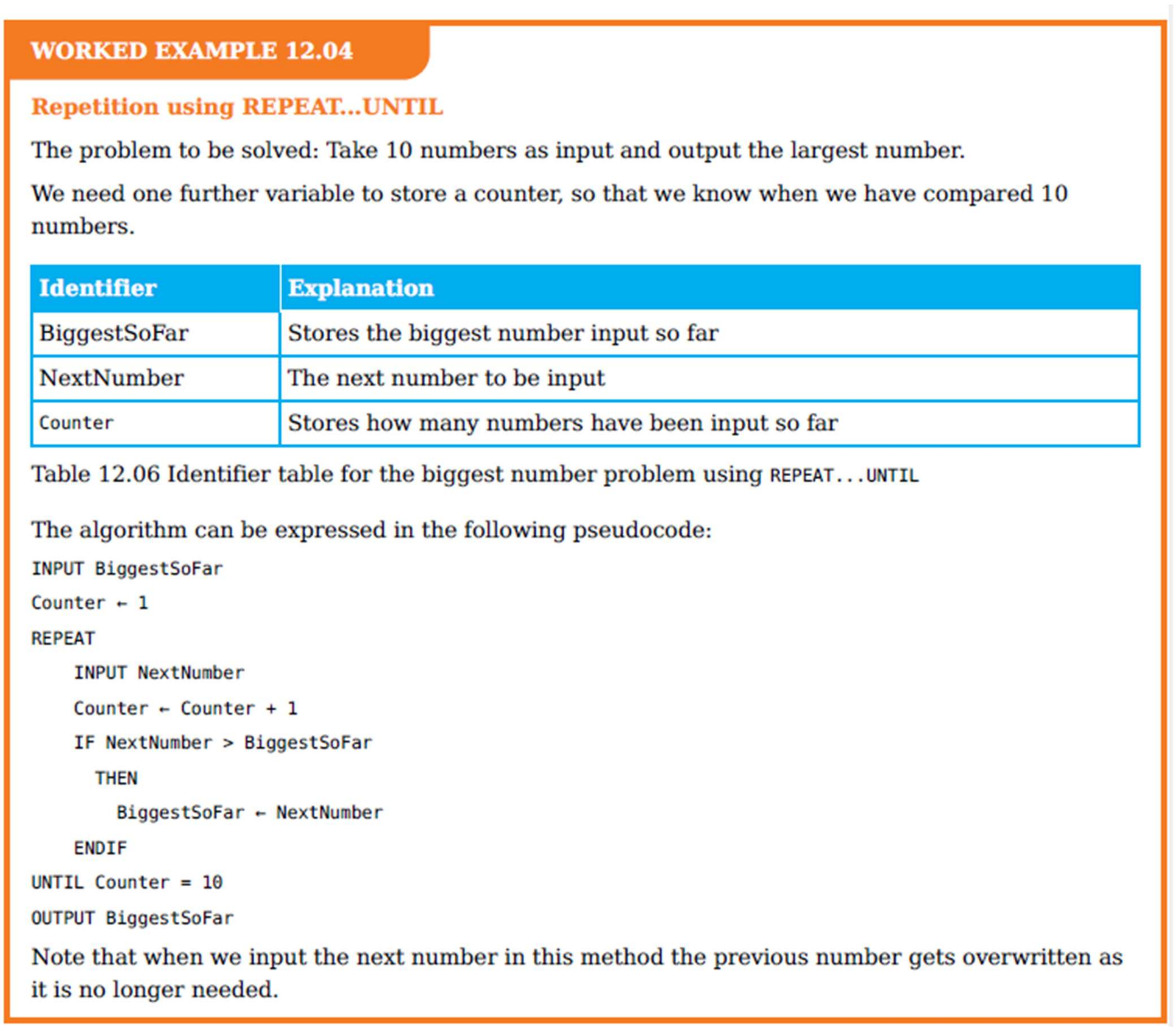
We need two variables to store the number of rows and the number of columns. We also need a variable to store the symbol. We need a counter for the rows and a counter for the columns.

Identifier	Explanation
`NumberOfRows`	Stores the number of rows of the grid
`NumberOfColumns`	Stores the number of columns of the grid
`Symbol`	Stores the chosen character symbol
`RowCounter`	Counts the number of rows
`ColumnCounter`	Counts the number of columns

Table 12.11 Identifier table for the nested loop example

```
INPUT NumberOfRows
INPUT NumberOfColumns
INPUT Symbol
FOR RowCounter ← 1 TO NumberOfRows
    FOR ColumnCounter ← 1 TO NumberOfColumns
        OUTPUT Symbol // without moving to next line
    NEXT ColumnCounter
    OUTPUT Newline   // move to the next line
NEXT RowCounter
```

Each time round the outer loop (counting the number of rows) we complete the inner loop, outputting a symbol for each count of the number of columns. This type of construct is called a **nested loop**.



WORKED EXAMPLE 12.04

Repetition using REPEAT...UNTIL

The problem to be solved: Take 10 numbers as input and output the largest number.

We need one further variable to store a counter, so that we know when we have compared 10 numbers.

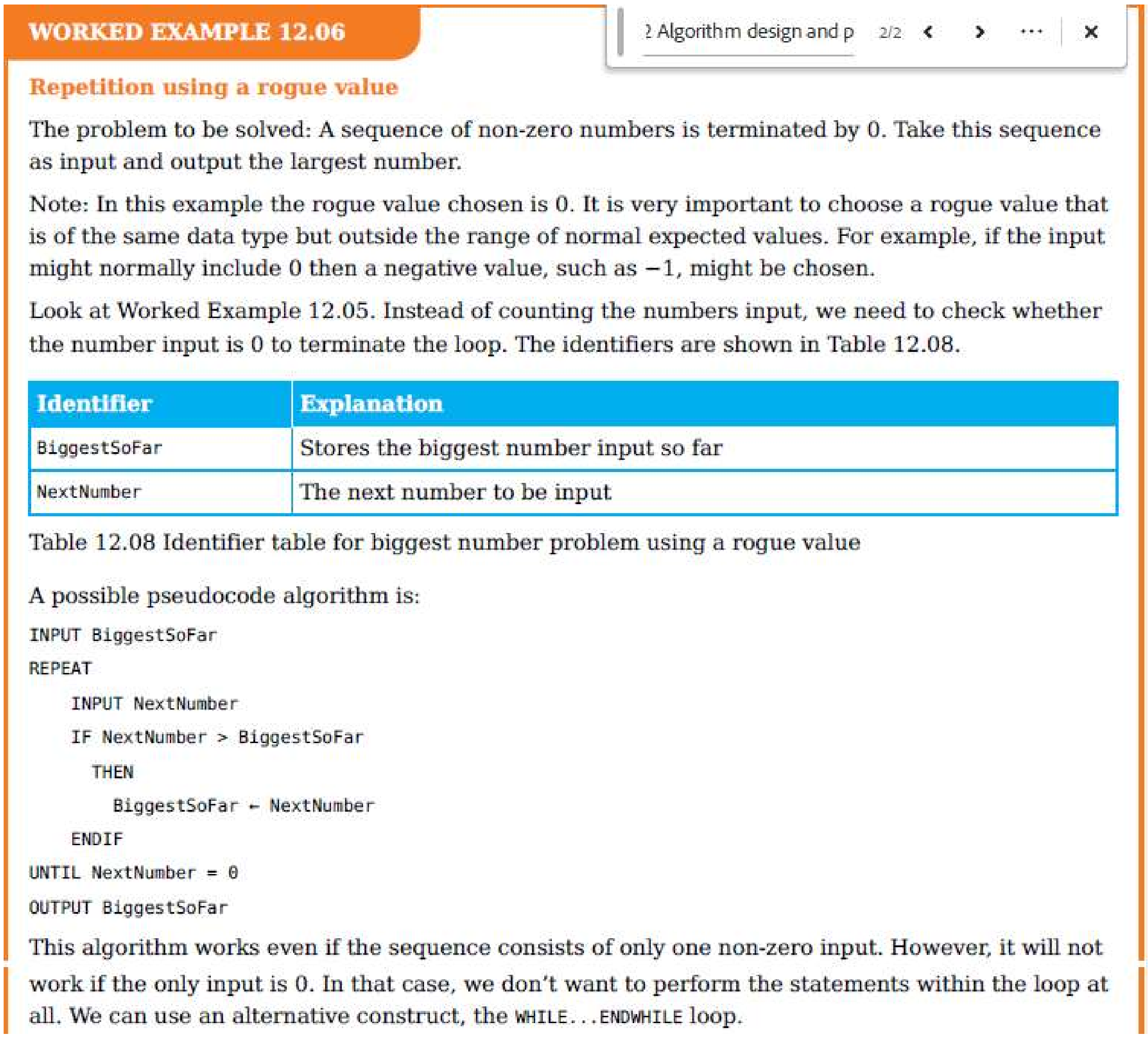
Identifier	Explanation
BiggestSoFar	Stores the biggest number input so far
NextNumber	The next number to be input
Counter	Stores how many numbers have been input so far

Table 12.06 Identifier table for the biggest number problem using REPEAT...UNTIL

The algorithm can be expressed in the following pseudocode:

```
INPUT BiggestSoFar
Counter ← 1
REPEAT
    INPUT NextNumber
    Counter ← Counter + 1
    IF NextNumber > BiggestSoFar
      THEN
        BiggestSoFar ← NextNumber
    ENDIF
UNTIL Counter = 10
OUTPUT BiggestSoFar
```

Note that when we input the next number in this method the previous number gets overwritten as it is no longer needed.


2 Algorithm design and p
2/2

WORKED EXAMPLE 12.06

Repetition using a rogue value

The problem to be solved: A sequence of non-zero numbers is terminated by 0. Take this sequence as input and output the largest number.

Note: In this example the rogue value chosen is 0. It is very important to choose a rogue value that is of the same data type but outside the range of normal expected values. For example, if the input might normally include 0 then a negative value, such as −1, might be chosen.

Look at Worked Example 12.05. Instead of counting the numbers input, we need to check whether the number input is 0 to terminate the loop. The identifiers are shown in Table 12.08.

Identifier	Explanation
`BiggestSoFar`	Stores the biggest number input so far
`NextNumber`	The next number to be input

Table 12.08 Identifier table for biggest number problem using a rogue value

A possible pseudocode algorithm is:

```
INPUT BiggestSoFar
REPEAT
    INPUT NextNumber
    IF NextNumber > BiggestSoFar
      THEN
        BiggestSoFar ← NextNumber
    ENDIF
UNTIL NextNumber = 0
OUTPUT BiggestSoFar
```

This algorithm works even if the sequence consists of only one non-zero input. However, it will not work if the only input is 0. In that case, we don't want to perform the statements within the loop at all. We can use an alternative construct, the `WHILE...ENDWHILE` loop.

WORKED EXAMPLE 12.07

Implementing the number-guessing game with a loop

Consider the number-guessing game again, this time allowing repeated guesses.

1. The player repeatedly inputs a number to guess the secret number stored.
2. If the guess is correct, the number of guesses made is output and the game stops.
3. If the number input is larger than the secret number, the player is given the message to input a smaller number.
4. If the number input is smaller than the secret number, the player is given the message to input a larger number.

The algorithm is expressed in structured English, as a flowchart and in pseudocode.

Algorithm for the number-guessing game in structured English:

```
SET value for secret number
REPEAT the following UNTIL correct guess
   INPUT guess
   count number of guesses
   COMPARE guess with secret number
   OUTPUT comment
OUTPUT number of guesses
```

We need variables to store the following values:

- the secret number (to be set as a random number)
- the number input by the player as a guess
- the count of how many guesses the player has made so far.

We represent this information in the identifier table shown in Table 12.09.

Identifier	Explanation
SecretNumber	The number to be guessed
NumberOfGuesses	The number of guesses the player has made
Guess	The number the player has input as a guess

Pseudocode for the number-guessing game with a post-condition loop

```
SecretNumber ← Random
NumberOfGuesses ← 0
REPEAT
    INPUT Guess
    NumberOfGuesses ← NumberOfGuesses + 1
    IF Guess > SecretNumber
      THEN
        // the player is given the message to input a smaller number
    ENDIF
    IF Guess < SecretNumber
      THEN
        // the player is given the message to input a larger number
    ENDIF
UNTIL Guess = SecretNumber
OUTPUT NumberOfGuesses
```

WORKED EXAMPLE 12.10

Drawing a pyramid using stepwise refinement

The problem to be solved: Take as input a chosen symbol and an odd number. Output a pyramid shape made up entirely of the chosen symbol, with the number of symbols in the final row matching the number input.

For example the two input values A and 9 result in the following output:

```
    A
   AAA
  AAAAA
 AAAAAAA
AAAAAAAAA
```

Identifier	Explanation
`Symbol`	The character symbol to form the pyramid
`MaxNumberOfSymbols`	The number of symbols in the final row
`NumberOfSpaces`	The number of spaces to be output in the current row
`NumberOfSymbols`	The number of symbols to be output in the current row

```
01      // Set Values
01.1    INPUT Symbol
01.2    // Input max number of symbols (an odd number)
01.2.1  REPEAT
01.2.2      INPUT MaxNumberOfSymbols
01.2.3  UNTIL MaxNumberOfSymbols MOD 2 = 1
01.3    NumberOfSpaces ← (MaxNumberOfSymbols – 1) / 2
01.4    NumberOfSymbols ← 1
02      REPEAT
03          // Output number of spaces
03.1        FOR i ← 1 TO NumberOfSpaces
03.2            OUTPUT Space // without moving to next line
03.3        NEXT i
04          // Output number of symbols
04.1        FOR i ← 1 TO NumberOfSymbols
04.2            OUTPUT Symbol // without moving to next line
04.3        NEXT i
04.4        OUTPUT Newline // move to the next line
05          // Adjust Values For Next Row
05.1        NumberOfSpaces ← NumberOfSpaces – 1
05.2        NumberOfSymbols ← NumberOfSymbols + 2
06      UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

Pre-condition loops

Pre-condition loops, as the name suggests, evaluate the condition before the statements within the loop are executed. Pre-condition loops will execute the statements within the loop as long as the condition evaluates to True. When the condition evaluates to False, execution will go to the next statement after the loop. Note that any variable used in the condition must not be undefined when the loop structure is first encountered.

When coding a pre-condition loop, you must ensure that there is a statement within the loop that will at some point change the value of the controlling condition. Otherwise the loop will execute forever.

In pseudocode the pre-condition loop is written as:

```
WHILE <condition> DO
    <statement(s)>
ENDWHILE
```

Pseudocode example,

```
Answer ← ""
WHILE Answer <> "Y" DO
    INPUT "Enter Y or N: " Answer
ENDWHILE
```

Calculate sum of 5 numbers.

```
DECLARE sum, i : INTEGER
sum ← 0
i ← 1
WHILE i <= 5 DO
        sum ← sum + i
        i ← i + 1
ENDWHILE
OUTPUT "The sum of 5 numbers is ", sum
```

Write pseudocode

1. input a value (all values will be positive integers)

2. count the number of odd values and count the number of even values

3. repeat from step 1 until the value input is 99

4. output the two count values, with a suitable message.

```
DECLARE COdd, CEven, ThisNum : INTEGER
COdd ← 0
CEven ← 0
INPUT ThisNum
WHILE ThisNum <> 99 DO
   IF ThisNum MOD 2 = 1 THEN
      COdd ← COdd + 1
   ELSE
      CEven ← CEven + 1
   ENDIF
   INPUT ThisNum
ENDWHILE
```

```
OUTPUT "Count of odd and even numbers: ", COdd, CEven
```

Using REPEAT UNTILL

```
DECLARE COdd, CEven, ThisNum : INTEGER
COdd ← 0
CEven ← 0

REPEAT
   INPUT ThisNum
   IF ThisNum <> 99 THEN
      IF ThisNum MOD 2 = 1 THEN
         COdd ← COdd + 1
      ELSE
         CEven ← CEven + 1
      ENDIF
   ENDIF
UNTIL ThisNum = 99

OUTPUT "Count of odd and even numbers: ", COdd, CEven
```

Which loop structure to use?

If you know how many times around the loop you need to go when the program execution gets to the loop statements, use a count-controlled loop. If the termination of the loop depends on some condition determined by what happens within the loop, then use a conditional loop. A pre-condition loop has the added benefit that the loop may not be entered at all, if the condition does not require it.

				A				
			A	A	A			
		A	A	A	A	A		
	A	A	A	A	A	A	A	
A	A	A	A	A	A	A	A	A

12.09 Modules

Another method of developing a solution is to decompose the problem into sub-tasks. Each sub-task can be considered as a 'module' that is refined separately. Modules are procedures and functions.

14.08 Procedures

A **procedure** groups together a number of steps and gives them a name (an identifier).

When we want to program a procedure we need to define it before the main program. We call it in the main program when we want the statements in the procedure body to be executed.

In pseudocode, a procedure definition is written as:

```
PROCEDURE <procedureIdentifier> // this is the procedure header
   <statement(s)>   // these statements are the procedure body
ENDPROCEDURE
```

This procedure is called using the pseudocode statement:

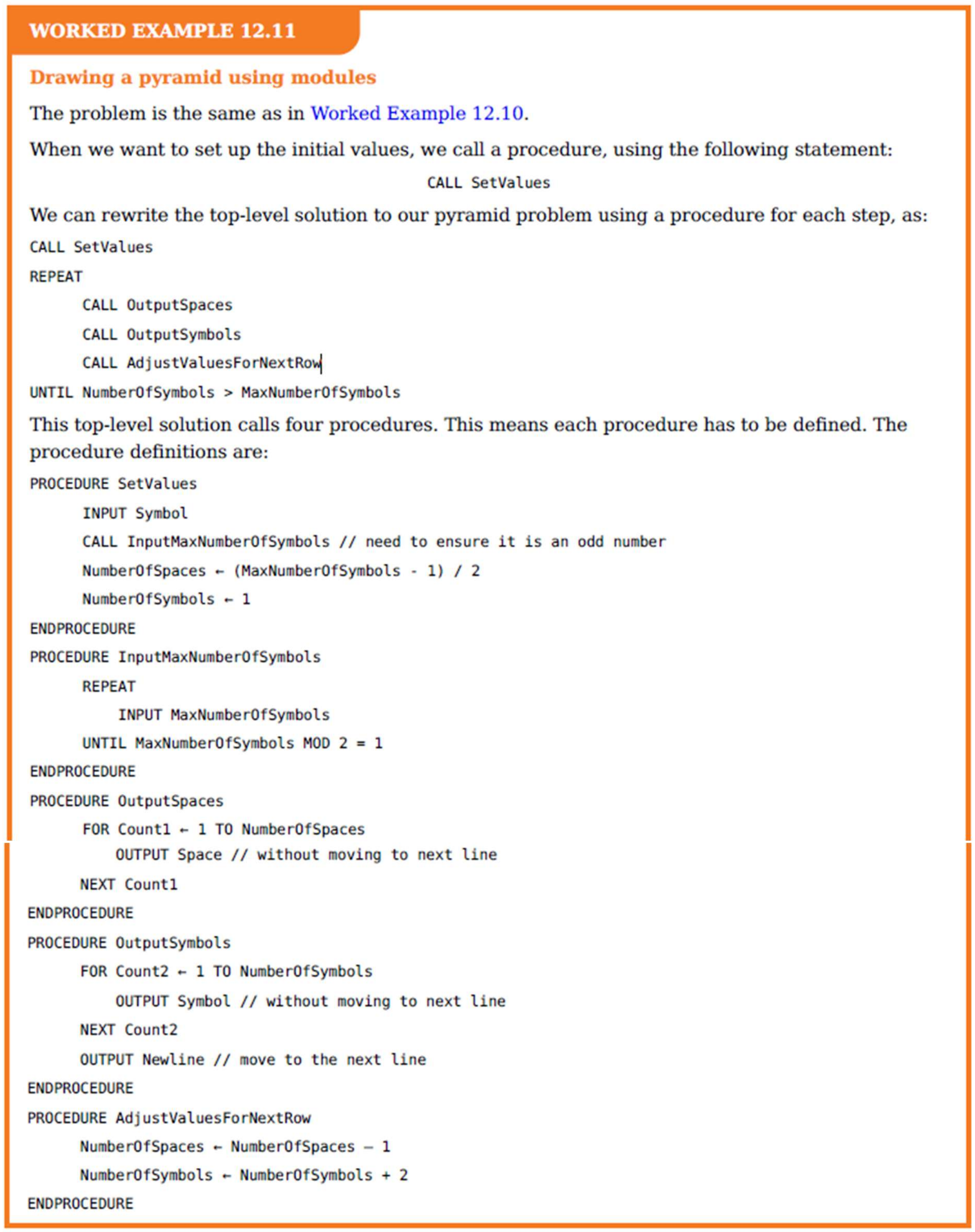
```
CALL <procedureIdentifier>
```

Here is an example pseudocode procedure definition:

```
PROCEDURE InputOddNumber
    REPEAT
        INPUT "Enter an odd number: " Number
    UNTIL Number MOD 2 = 1
    OUTPUT "Valid number entered"
ENDPROCEDURE
```

This procedure is called using the CALL statement:

```
CALL InputOddNumber
```



WORKED EXAMPLE 12.11

Drawing a pyramid using modules

The problem is the same as in Worked Example 12.10.

When we want to set up the initial values, we call a procedure, using the following statement:

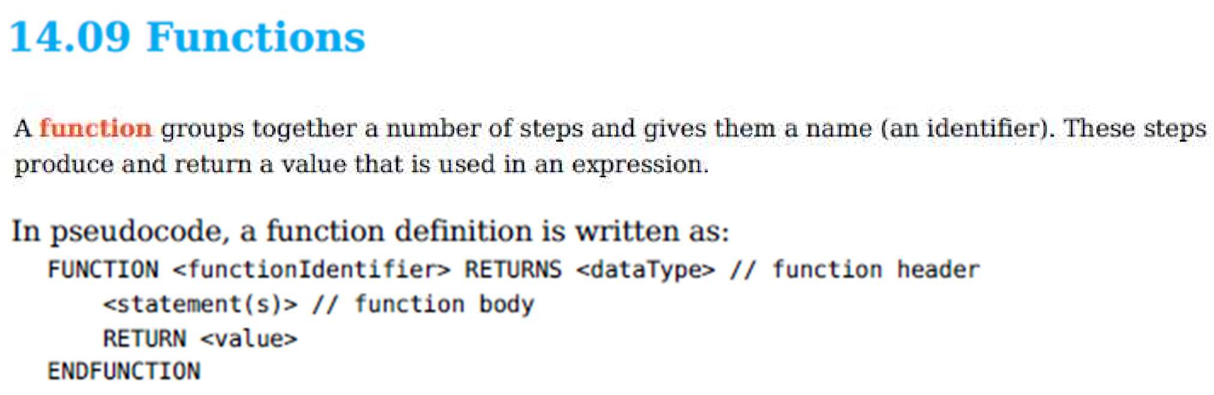
```
CALL SetValues
```

We can rewrite the top-level solution to our pyramid problem using a procedure for each step, as:

```
CALL SetValues
REPEAT
    CALL OutputSpaces
    CALL OutputSymbols
    CALL AdjustValuesForNextRow
UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

This top-level solution calls four procedures. This means each procedure has to be defined. The procedure definitions are:

```
PROCEDURE SetValues
    INPUT Symbol
    CALL InputMaxNumberOfSymbols // need to ensure it is an odd number
    NumberOfSpaces ← (MaxNumberOfSymbols - 1) / 2
    NumberOfSymbols ← 1
ENDPROCEDURE
PROCEDURE InputMaxNumberOfSymbols
    REPEAT
        INPUT MaxNumberOfSymbols
    UNTIL MaxNumberOfSymbols MOD 2 = 1
ENDPROCEDURE
PROCEDURE OutputSpaces
    FOR Count1 ← 1 TO NumberOfSpaces
        OUTPUT Space // without moving to next line
    NEXT Count1
ENDPROCEDURE
PROCEDURE OutputSymbols
    FOR Count2 ← 1 TO NumberOfSymbols
        OUTPUT Symbol // without moving to next line
    NEXT Count2
    OUTPUT Newline // move to the next line
ENDPROCEDURE
PROCEDURE AdjustValuesForNextRow
    NumberOfSpaces ← NumberOfSpaces – 1
    NumberOfSymbols ← NumberOfSymbols + 2
ENDPROCEDURE
```

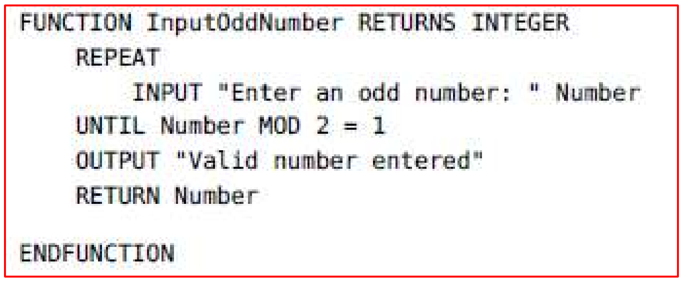


14.09 Functions

A **function** groups together a number of steps and gives them a name (an identifier). These steps produce and return a value that is used in an expression.

In pseudocode, a function definition is written as:

```
FUNCTION <functionIdentifier> RETURNS <dataType> // function header
    <statement(s)> // function body
    RETURN <value>
ENDFUNCTION
```



```
FUNCTION InputOddNumber RETURNS INTEGER
    REPEAT
        INPUT "Enter an odd number: " Number
    UNTIL Number MOD 2 = 1
    OUTPUT "Valid number entered"
    RETURN Number

ENDFUNCTION
```

Write a function to find the sum of two numbers.

```
FUNCTION AddNumbers (a, b) RETURNS INTEGER
        RETURN a+b
ENDFUNCTION

result = AddNumber(5,6)
OUTPUT “The Sum is”, result
```

Write a procedure that takes two numbers as input and prints their sum.

```
PROCEDURE DisplaySum(a, b)
   DECLARE sum : INTEGER
   sum = a + b
   OUTPUT  "The sum is:", sum
END PROCEDURE

   CALL DisplaySum(5, 6)
```

WORKED EXAMPLE 12.12

Drawing a pyramid using modules

The problem is the same as in Worked Example 12.11.

We can rewrite the top-level solution to our pyramid problem using procedures and functions.

```
01   CALL SetValues
02   REPEAT
03        CALL OutputSpaces
04        CALL OutputSymbols
05.1      NumberOfSpaces ← AdjustedNumberOfSpaces
05.2      NumberOfSymbols ← AdjustedNumbeOfSymbols
06   UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

This top-level solution calls three procedures. It also makes use of two functions in lines 05.1 and 05.2.

The procedures and functions have to be defined.

```
PROCEDURE SetValues
    INPUT Symbol
    MaxNumberOfSymbols ← ValidatedMaxNumberOfSymbols
    NumberOfSpaces ← (MaxNumberOfSymbols - 1) / 2
    NumberOfSymbols ← 1
ENDPROCEDURE
FUNCTION ValidatedMaxNumberOfSymbols RETURNS INTEGER
    REPEAT
        INPUT MaxNumberOfSymbols
    UNTIL MaxNumberOfSymbols MOD 2 = 1
    RETURN MaxNumberOfSymbols
ENDFUNCTION
PROCEDURE OutputSpaces
    FOR Count1 ← 1 TO NumberOfSpaces
        OUTPUT Space // without moving to next line
    NEXT Count1
ENDPROCEDURE
PROCEDURE OutputSymbols
    FOR Count2 ← 1 TO NumberOfSymbols
       OUTPUT Symbol // without moving to next line
    NEXT Count2
    OUTPUT Newline // move to the next line
ENDPROCEDURE
FUNCTION AdjustedNumberOfSpaces RETURNS INTEGER
    NumberOfSpaces ← NumberOfSpaces – 1
    RETURN NumberOfSpaces
ENDFUNCTION
FUNCTION AdjustedNumberOfSymbols RETURNS INTEGER
    NumberOfSymbols ← NumberOfSymbols + 2
    RETURN NumberOfSymbols
ENDFUNCTION
```

Note that procedure `SetValues` uses a function `ValidatedMaxNumberOfSymbols`.

14.11 Passing parameters to functions

The **function header** is written in pseudocode as:

```
FUNCTION <functionIdentifier> (<parameterList>) RETURNS <dataType>
```

where `<parameterList>` is a list of identifiers and their data types, separated by commas.

Here is an example pseudocode function definition that uses parameters:

```
FUNCTION SumRange(FirstValue : INTEGER, LastValue : INTEGER) RETURNS INTEGER
    DECLARE Sum, ThisValue : INTEGER
    Sum ← 0
    FOR ThisValue ← FirstValue TO LastValue
        Sum ← Sum + ThisValue
    NEXT ThisValue
    RETURN Sum
ENDFUNCTION
```

14.12 Passing parameters to procedures

If a parameter is passed **by value**, at call time the argument can be an actual value (as we showed in the code examples in Section 14.11). If the argument is a variable, then a copy of the current value of the variable is passed into the subroutine. The value of the variable in the calling program is not affected by what happens in the subroutine.

For procedures, a parameter can be passed **by reference**. At call time, the argument must be a variable. A pointer to the memory location (the address) of that variable is passed into the procedure. Any changes that are applied to the variable's contents will be effective outside the procedure in the calling program/module.

Note that neither of these methods of parameter passing applies to Python. In Python or Java, the method is called pass by object reference. This is basically an object-oriented way of passing parameters and is beyond the scope of this chapter (objects are dealt with in Chapter 27). The important point is to understand how to program in Python and Java to get the desired effect.

The full **procedure header** is written in pseudocode, in a very similar fashion to that for function headers, as:

```
PROCEDURE <ProcedureIdentifier> (<parameterList>)
```

The parameter list needs more information for a procedure definition. In pseudocode, a parameter in the list is represented in one of the following formats:

```
BYREF <identifier1> : <dataType>
BYVALUE <identifier2> : <dataType>
```

Passing parameters by value

The pseudocode for the pyramid example in Chapter 12 (Section 12.09) includes a procedure definition that uses two parameters passed by value. We can now make that explicit:

```
PROCEDURE OutputSymbols(BYVALUE NumberOfSymbols : INTEGER, Symbol : CHAR)
    DECLARE Count : INTEGER
    FOR Count ← 1 TO NumberOfSymbols
        OUTPUT Symbol // without moving to next line
    NEXT Count
    OUTPUT NewLine
ENDPROCEDURE
```

Passing parameters by reference

When parameters are passed by reference, when the values inside the subroutine change, this affects the values of the variables in the calling program.

Consider the pseudocode procedure AdjustValuesForNextRow below.

The pseudocode for the pyramid example generated in Chapter 12 (Section 12.09) includes a procedure definition that uses two parameters passed by reference. We can now make that explicit:

```
PROCEDURE AdjustValuesForNextRow(BYREF Spaces : INTEGER, Symbols : INTEGER)
    Spaces ← Spaces - 1
    Symbols ← Symbols + 2
ENDPROCEDURE
```

The pseudocode statement to call the procedure is:

```
CALL AdjustValuesForNextRow(NumberOfSpaces, NumberOfSymbols)
```

The values of the parameters `Spaces` and `Symbols` are changed within the procedure when this is called. The variables NumberOfSpaces and `NumberOfSymbols` in the program code after the call will store the updated values that were passed back from the procedure.

WORKED EXAMPLE 12.13

Drawing a pyramid using modules and parameters

The problem is the same as in Worked Example 12.12.

```
CALL SetValues(Symbol, MaxNumberOfSymbols, NumberOfSpaces, NumberOfSymbols)
REPEAT
    CALL OutputSpaces(NumberOfSpaces)
    CALL OutputSymbols(NumberOfSymbols, Symbol)
    NumberOfSpaces ← AdjustedNumberOfSpaces(NumberOfSpaces)
    NumberOfSymbols ← AdjustedNumbeOfSymbols(NumberOfSymbols)
UNTIL NumberOfSymbols > MaxNumberOfSymbols
```

Module definitions:

```
PROCEDURE SetValues(Symbol, MaxNumberOfSymbols, NumberOfSpaces, NumberOfSymbols)
    INPUT Symbol
    MaxNumberOfSymbols ← ValidatedMaxNumberOfSymbols
    NumberOfSpaces ← (MaxNumberOfSymbols - 1) / 2
    NumberOfSymbols ← 1
ENDPROCEDURE
FUNCTION ValidatedMaxNumberOfSymbols RETURNS INTEGER
    REPEAT
        INPUT MaxNumberOfSymbols
    UNTIL MaxNumberOfSymbols MOD 2 = 1
    RETURN MaxNumberOfSymbols
ENDFUNCTION
PROCEDURE OutputSpaces(NumberOfSpaces)
    FOR Count1 ← 1 TO NumberOfSpaces
        OUTPUT Space // without moving to next line
    NEXT Count1
ENDPROCEDURE
PROCEDURE OutputSymbols(NumberOfSymbols, Symbol)
    FOR Count2 ← 1 TO NumberOfSymbols
        OUTPUT Symbol // without moving to next line
    NEXT Count2
    OUTPUT Newline // move to the next line
ENDPROCEDURE
FUNCTION AdjustedNumberOfSpaces(NumberOfSpaces) RETURNS INTEGER
    NumberOfSpaces ← NumberOfSpaces – 1
    RETURN NumberOfSpaces
ENDFUNCTION
FUNCTION AdjustedNumbeOfSymbols(NumberOfSymbols) RETURNS INTEGER
    NumberOfSymbols ← NumberOfSymbols + 2
    RETURN NumberOfSymbols
ENDFUNCTION
```

(b) The following is a pseudocode function.

Line numbers are given for reference only.

```
01  FUNCTION StringClean(InString : STRING) RETURNS STRING
02
03     DECLARE NextChar : CHAR
04     DECLARE OutString : STRING
05     DECLARE Counter : INTEGER
06
07     OutString ← ""
08
09     FOR Counter ← 1 TO LENGTH(InString)
10        NextChar ← MID(InString, Counter, 1)
11        NextChar ← LCASE(NextChar)
12        IF NOT((NextChar < 'a') OR (NextChar > 'z')) THEN
13           OutString ← OutString & NextChar
14        ENDIF
15     NEXT Counter
16
17     RETURN OutString
18
19  ENDFUNCTION
```

(i) Examine the pseudocode and complete the following table.

	Answer
Give a line number containing an example of an initialisation statement.	
Give a line number containing the start of a repeating block of code.	
Give a line number containing a logic operation.	
Give the number of parameters to the function `MID()`.	

[4]

(ii) Write a simplified version of the statement in line `12`.

....

.... [2]

6 A procedure `CountVowels()` will:

- be called with a string containing alphanumeric characters as its parameter
- count and output the number of occurrences of each vowel (a, e, i, o, u) in the string
- count and output the number of occurrences of the other alphabetic characters (as a single total).

The string may contain both upper and lower case characters.

Each count value will be stored in a unique element of a global 1D array `CharCount` of type `INTEGER`. The array will contain six elements.

Write pseudocode for the procedure `CountVowels()`.

```
PROCEDURE CountVowels(ThisString : STRING)

  DECLARE Index : INTEGER
  DECLARE ThisChar : CHAR

  FOR Index ← 1 to 6
     CharCount[Index] ← 0 //initialise elements
  NEXT Index
  Index ← 1

  FOR Index ← 1 TO LENGTH(ThisString)
     ThisChar ← LCASE(MID(ThisString, Index, 1))

     CASE OF ThisChar
        'a'      : CharCount[1] ← CharCount[1] + 1
        'e'      : CharCount[2] ← CharCount[2] + 1
        'i'      : CharCount[3] ← CharCount[3] + 1
        'o'      : CharCount[4] ← CharCount[4] + 1
        'u'      : CharCount[5] ← CharCount[5] + 1
        'a' TO 'z': CharCount[6] ← CharCount[6] + 1
     ENDCASE
  NEXT Index

  FOR Index ← 1 to 6
     OUTPUT CharCount[Index] //output results
  NEXT Index

ENDPROCEDURE
```

1 mark for each of the following:

1 Procedure heading (with parameter) and ending
2 Declare local variable for `Index` as loop counter but **not** `CharCount` array
3 Initialise elements of `CharCount` array to zero
4 Loop through all characters in `ThisString`
5 Use of `MID()` to extract single character
6 Test for each vowel **and** increment associated count
7 Test for consonants **and** increment associated count
8 Output the results (supporting text not necessary) **after the loop**

Note: Max 7 if `CharCount` not used to store count