

Programming

AS Computer Science ■ Topic 11

A-Level Computer Science



What we will cover

11.1 Variables, constants, input, output

11.1.1 Operators and expressions

11.1.2 Built-in functions

11.2 IF and nested IF

11.2.1 CASE

11.2.2 FOR loops: totals and counts

11.2.3 WHILE and REPEAT; choosing a loop

11.2.4 Trace tables

11.3 Procedures

11.3.1 Functions; procedure or function?

11.3.2 BYVAL, BYREF and scope

11.3.3 Efficient pseudocode and modules

Each part of this deck has the number of its exercise sheet and past-paper sheet. The last part goes beyond the sheets.

1

Sheet 11.1 – Variables and constants

Sheet 11.1 – what you need to know

You must be able to:

- declare variables and constants with suitable types
- write assignment, input and output statements
- document variables in an identifier table

What the exam asks

“Complete the table by identifying the test method that matches the test description.” [3]

“Describe the benefits of your answer with reference to this program.” [3]

“State a suitable way of documenting the variables and give one piece of information to record.” [2]

Practise on exercise sheet 11.1, then try the real exam questions on past-paper sheet 11.1.

Constants and variables

A **constant** 常量 holds a value that **never changes**; a variable holds one that may change. Declare them with a type:

```
CONSTANT Pi = 3.14159  
DECLARE Radius : REAL  
DECLARE Area : REAL
```

```
Radius ← 5
```

```
Area ← Pi * Radius * Radius
```

Count

7 → 8

variable: can change

Pi

3.14159

constant: fixed

Use constants for fixed values that recur (Pi, MaxScore): they make code clearer and easy to change in one place.

Why use a constant: the exam answer

A constant answers “identify a more appropriate way of representing” a fixed value, such as a tax rate or a maximum score, that appears at several places in the pseudocode.

1

the value is set once and **cannot be changed accidentally**

2

a change is made **in one place** and reaches every statement that uses it

3

the identifier gives the value a **meaning**: MaxScore, not 100

4

less risk of a **typing error** in a long value such as 3.14159

“State a value that could be replaced by a constant” wants the **literal** from the pseudocode (0.2, 40), not a new name.

The six data types

Declare each variable before use: an **identifier** 标识符 (its name) and a data type.

Type	Holds	Written as	Typical use
INTEGER	whole numbers	42, -3	a count, an index, a loop counter
REAL	numbers with a fractional part	3.75	a price, an average
CHAR	one character	'A' (single quotes)	a grade letter, a menu key
STRING	a sequence of characters	"Hello" (double quotes)	a name, a postcode
BOOLEAN	TRUE or FALSE	TRUE	a flag such as Found
DATE	a calendar date	12/05/2026	a date of birth

Worked example: choose the data type

Answer from how the variable is **used**, and write the type in capitals, spelled as the guide spells it.

Statement	Type	Because
Found ← FALSE	BOOLEAN	it is set to TRUE or FALSE
Initial ← 'K'	CHAR	one character, in single quotes
Price ← 12.99	REAL	it has a decimal point
Count ← Count + 1	INTEGER	a whole number that goes up by one
Half ← Total DIV 2	INTEGER	used with DIV, like anything used with MOD or as an array index
Name ← "Li Wei"	STRING	text in double quotes

Assignment, input and output, operators

Use $\leftarrow$ for **assignment** 赋值:

```
Total  $\leftarrow$  Total + 1  
Average  $\leftarrow$  Sum / Count
```

Input and output:

```
OUTPUT "Enter your name:"  
INPUT Name  
OUTPUT "Hello, ", Name
```

Expressions use **operators** 运算符:

- arithmetic + - * /, plus **DIV** and **MOD**:
7 **DIV** 2 = 3, 7 **MOD** 2 = 1
- comparisons = <> < > <= >=
- logic **AND** **OR** **NOT**

Precedence 优先级, highest to lowest:

NOT $\rightarrow$ * / **DIV** **MOD** $\rightarrow$ + - $\rightarrow$ comparisons
 $\rightarrow$ **AND** $\rightarrow$ **OR**

Use brackets when unsure.

x $\leftarrow$ 5 **stores** 5 in x; x = 5 **asks** whether x holds 5.

Following assignments: the swap

```
A ← 3  
B ← 8  
Temp ← A  
A ← B  
B ← Temp
```

after the statement	A	B	Temp
A ← 3	3		
B ← 8	3	8	
Temp ← A	3	8	3
A ← B	8	8	3
B ← Temp	8	3	3

The arrow means “becomes”: the **right** side is worked out first, then stored on the **left**. A swap needs a third variable, because $A \leftarrow B$ destroys the old value of A.

2

Sheet 11.1.1 – Operators and expressions

Sheet 11.1.1 – what you need to know

You must be able to:

- evaluate expressions with DIV, MOD and precedence
- give the data type of a result
- evaluate Boolean expressions

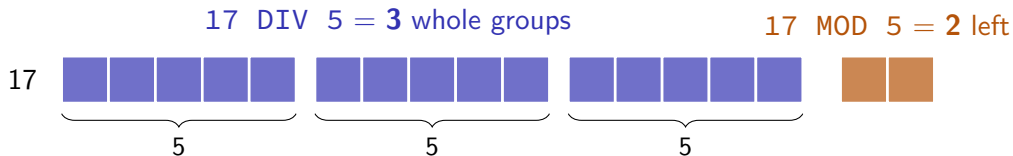
What the exam asks

“Describe the error in each case, or write ‘NO ERROR.’” [5]

“Complete the table by evaluating each expression.” [4]

Practise on exercise sheet 11.1.1, then try the real exam questions on past-paper sheet 11.1.

DIV and MOD



the job	the expression
is N even?	$N \text{ MOD } 2 = 0$
the last digit of N	$N \text{ MOD } 10$
N without its last digit	$N \text{ DIV } 10$
135 minutes in hours	$135 \text{ DIV } 60 \rightarrow 2$
...and the minutes left	$135 \text{ MOD } 60 \rightarrow 15$

/ always gives a **REAL** (8 / 2 is 4.0). DIV and MOD give **INTEGERS**.

Precedence: which operator goes first?

1 brackets

2 * / DIV MOD

3 + -

4 comparisons: = <> < > <= >=

5 NOT, then AND, then OR

$3 + 4 * 2 \rightarrow 3 + 8 \rightarrow 11$

$(3 + 4) * 2 \rightarrow 7 * 2 \rightarrow 14$

$20 - 17 \text{ MOD } 5 * 2 \rightarrow 20 - 2 * 2 \rightarrow 16$

A comparison gives a **BOOLEAN**: $12 > 4$ is TRUE.

"12" + 3 is an **ERROR**: a string cannot be added to a number.

Mean $\leftarrow A + B + C / 3$ divides only C. Write $(A + B + C) / 3$.

Worked example: Boolean expressions

With Points $\leftarrow$ 100, Active $\leftarrow$ TRUE and Exempt $\leftarrow$ FALSE:

Expression	Value	Why
(Points > 99) OR Active	TRUE	both are true; one would do
(Points MOD 2 = 0) OR Exempt	TRUE	100 MOD 2 is 0
(Points <= 75) AND (Active OR Exempt)	FALSE	the first side is false
(Active OR NOT Active) AND NOT Exempt	TRUE	first bracket always true

X OR NOT X is TRUE whatever X is, so the last expression is just NOT Exempt. Evaluate brackets first, then NOT, then AND, then OR.

3

Sheet 11.1.2 – Built-in functions

Sheet 11.1.2 – what you need to know

You must be able to:

- use the functions in the Paper 2 insert, one inside another
- use RAND, INT, ASC, CHR and the date routines
- use an IDE to find errors

What the exam asks

“Describe the ways in which Jennifer can use both a compiler and an interpreter.” [4]

“Describe how a typical IDE could be used to help debug the program.” [4]

“Describe the error in each case, or write ‘NO ERROR.’” [5]

Practise on exercise sheet 11.1.2, then try the real exam questions on past-paper sheet 11.1.2.

Library routines and the Paper 2 insert

Many tasks have ready-made **library routines** 库例程. The Paper 2 **insert** 附页 lists the ones you may use, with their exact names, parameters and return types; any other function a question needs is given in the question.

Use the 9618 names. The IGCSE names (UCASE, VAL, STR) are not accepted.

A **program library** 程序库 holds routines already written, compiled and tested.

“State three benefits”

- already **tested**, so fewer errors
- they **save development time**
- they do what you could not write
- written by experts, **reused** widely
- a fixed interface: call it anywhere

String routines

$s =$

1	2	3	4	5	6	7	8
C	O	M	P	U	T	E	R

 positions count from 1, not 0

Routine	Returns	Example, with s above
<code>LENGTH(s)</code>	the number of characters in s	<code>LENGTH(s)</code> = 8
<code>LEFT(s, n)</code>	the first n characters	<code>LEFT(s, 3)</code> = "COM"
<code>RIGHT(s, n)</code>	the last n characters	<code>RIGHT(s, 2)</code> = "ER"
<code>MID(s, i, n)</code>	n characters from position i	<code>MID(s, 4, 3)</code> = "PUT"

Case, joining and conversion routines

Routine	Returns	Example
<code>TO_UPPER(s)</code>	s in capital letters	<code>TO_UPPER("ab1") = "AB1"</code>
<code>TO_LOWER(s)</code>	s in small letters	<code>TO_LOWER("Hi") = "hi"</code>
<code>s1 & s2</code>	the two strings joined: concatenation 连接	<code>"A" & "BC" = "ABC"</code>
<code>NUM_TO_STR(x)</code>	the number x as a string	<code>NUM_TO_STR(7) = "7"</code>
<code>STR_TO_NUM(s)</code>	the string s as a number	<code>STR_TO_NUM("3.5") = 3.5</code>
<code>IS_NUM(s)</code>	TRUE if s is a valid number	<code>IS_NUM("12a") = FALSE</code>
<code>ASC(c)</code>	the character code of c	<code>ASC('A') = 65</code>
<code>CHR(n)</code>	the character with code n	<code>CHR(66) = 'B'</code>

Number, date and file routines

Routine	Returns
<code>INT(x)</code>	the whole-number part of x: <code>INT(7.9) = 7</code>
<code>RAND(n)</code>	a random real number from 0 up to, but not including, n
<code>DAY(d)</code> , <code>MONTH(d)</code> , <code>YEAR(d)</code>	the day, the month or the year of the date d
<code>DAYINDEX(d)</code>	the day of the week of d, where Sunday is 1
<code>SETDATE(d, m, y)</code>	a DATE built from three integers
<code>TODAY()</code>	today's date: <code>YEAR(TODAY())</code> is this year
<code>EOF(f)</code>	TRUE when the file f has no more lines to read

Dates and random numbers in one line

Statement	What it does
<code>SETDATE(17, 11, 2007)</code>	builds the date 17 November 2007
<code>12 - MONTH(MyDOB)</code>	the months from the month of birth to the end of the year
<code>IF DAYINDEX(MyDOB) = 5 THEN</code>	tests for a Thursday, because Sunday is day 1
<code>INT(RAND(High - Low + 1)) + Low</code>	a random integer from Low to High inclusive
<code>INT(RAND(21)) - 10</code>	a random integer from -10 to 10

`RAND(n)` returns a real number from 0 up to, but not including, n, so an integer always needs `INT`.

Worked example: evaluate each expression

Given Word $\leftarrow$ "Program", Code $\leftarrow$ 'Q' and N $\leftarrow$ 7:

Expression	Value	Why
LENGTH(Word)	7	seven characters
MID(Word, 4, 2)	"gr"	two characters, starting at position 4
LEFT(Word, 3) & "!"	"Pro!"	joined with &
TO_UPPER(RIGHT(Word, 2))	"AM"	the inner function runs first
ASC(Code) - ASC('A')	16	'Q' is 81 and 'A' is 65
N DIV 2 + N MOD 2	4	3 + 1
NUM_TO_STR(N) & "th"	"7th"	the number becomes a string first
INT(N / 2)	3	3.5 cut to its whole part

Work from the **inside out**, and keep the quotes: "7" is a string and 7 is a number.

Worked example: find the error

Describe the error in each statement, or write NO ERROR:

Statement	Error
Result $\leftarrow$ 2 & 4	& joins strings; numbers need +
SubString $\leftarrow$ MID("pseudocode", 4, 1)	NO ERROR: one character, "u"
IF x = 3 OR 4 THEN	OR needs x = 3 OR x = 4
Result $\leftarrow$ Status AND INT(x / 2)	INT(x / 2) is not a Boolean
Message $\leftarrow$ "Done" + LENGTH(MyString)	join with & after NUM_TO_STR

& works on strings; + - * / DIV MOD on numbers; AND OR NOT on Booleans; = and <> compare two values of the same type.

4

Sheet 11.2 – IF and nested IF

Sheet 11.2 – what you need to know

You must be able to:

- write IF and nested IF with the tests in the right order
- test at the boundaries
- name statements that can cause a run-time error

What the exam asks

“State the type of loop structure that would be the most appropriate and justify why.” [2]

“Write the three statements and explain how each could generate a run-time error.” [3]

“Write pseudocode using a single selection construct without using a CASE structure.” [2]

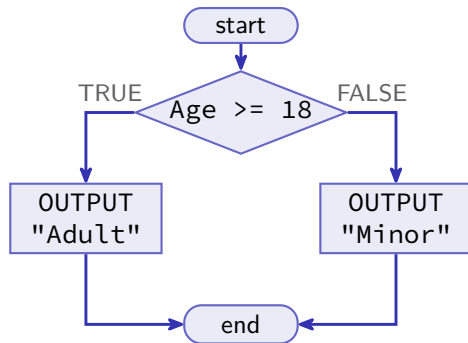
Practise on exercise sheet 11.2, then try the real exam questions on past-paper sheet 11.2.

Selection: IF, THEN, ELSE

Selection 选择 chooses which steps run.

```
IF Age >= 18 THEN  
    OUTPUT "Adult"  
ELSE  
    OUTPUT "Minor"  
ENDIF
```

The condition is tested **once**; exactly one branch runs.



Nested IF: the order of the tests matters

```
IF Mark >= 70 THEN
    OUTPUT "Merit"
ELSE
    IF Mark >= 50 THEN
        OUTPUT "Pass"
    ELSE
        OUTPUT "Fail"
    ENDIF
ENDIF
```

- the **first** condition that is TRUE wins:
test the **most demanding** condition first
- the inner IF is reached only when Mark >= 70 is FALSE, so Mark >= 50 alone is enough
- two IFs need two ENDIFs
- with Mark >= 50 tested first, a mark of 85 gives “Pass” and “Merit” can never be output: a **logic error** 逻辑错误

Boundaries and logic errors

Boundaries are where marks are lost.



“a mark of 50 or more passes”

Mark $\geq$ 50



the same, mistyped

Mark > 50

- the last **CASE** branch, for “anything else”, is **OTHERWISE**, not a condition such as > 200
- a wrong comparison is a **logic error** 逻辑错误: the program runs, but gives the wrong output for some inputs
- a trace table with a **boundary value** such as 50 is how you find it

A counter that rolls over, and run-time errors

```
PROCEDURE Tick()  
  Seconds ← Seconds + 1  
  IF Seconds = 60 THEN  
    Seconds ← 0  
    Minutes ← Minutes + 1  
    IF Minutes = 60 THEN  
      Minutes ← 0  
      Hours ← (Hours + 1) MOD 24  
    ENDIF  
  ENDIF  
ENDPROCEDURE
```

Each IF runs only when the unit below it has just rolled over: 23:59:59 becomes 00:00:00.

“Write three statements and explain how each could generate a run-time error.” [3]

Average ← Total / Count:
division by zero when Count = 0.
Name ← List[Index]: Index
outside the array's bounds.
READFILE "Data.txt", Line:
reading past the end of the file.

A run-time error appears only **while the program runs**, with some data; it compiles cleanly.

5

Sheet 11.2.1 – CASE

Sheet 11.2.1 – what you need to know

You must be able to:

- write CASE with ranges and OTHERWISE
- say when CASE cannot be used
- convert between CASE and IF

What the exam asks

“Outline, using stepwise refinement, the five steps for this algorithm.” [5]

“Identify this assignment and explain why this is the case.” [3]

“Give the appropriate data types for the variables ThisValue, Check and Level.” [3]

Practise on exercise sheet 11.2.1, then try the real exam questions on past-paper sheet 11.2.1.

Nested IF and CASE

For more than two cases, a **nested** 嵌套 **IF** works, but deep nesting is hard to read.

Nested IF: each **IF** needs its own **ENDIF**

```
IF Mark >= 50 THEN
    IF Mark >= 80 THEN
        OUTPUT "Distinction"
    ELSE
        OUTPUT "Pass"
    ENDIF
ELSE
    OUTPUT "Fail"
ENDIF
```

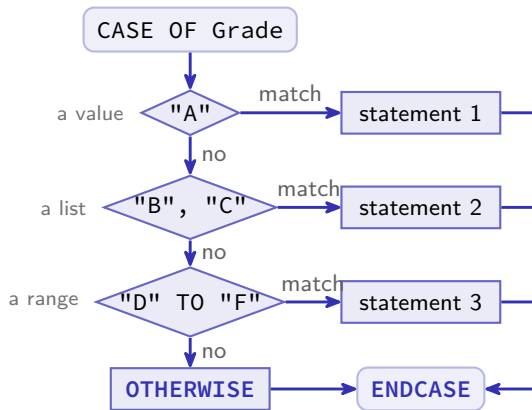
CASE: one value against several options

```
CASE OF Grade
    "A" : OUTPUT "Excellent"
    "B" : OUTPUT "Good"
    OTHERWISE : OUTPUT "Try again"
ENDCASE
```

A Cambridge **CASE** allows single values, value lists (1, 2, 3 :) and ranges (1 TO 5 :).

How a CASE statement runs

- the value is tested against each option **in order**
- the **first** option that matches runs its statement
- if none matches, **OTHERWISE** runs
- every branch rejoins at **ENDCASE**



Worked example: rewrite without CASE

The same functionality, without a **CASE** structure:

```
CASE OF MySwitch
  1 : ThisChar ← 'a'
  2 : ThisChar ← 'y'
  3 : ThisChar ← '7'
  OTHERWISE : ThisChar ← '*'
ENDCASE
```

Each value becomes a branch of a **chain of IFs**, and **OTHERWISE** becomes the **last ELSE**.

```
IF MySwitch = 1 THEN
  ThisChar ← 'a'
ELSE
  IF MySwitch = 2 THEN
    ThisChar ← 'y'
  ELSE
    IF MySwitch = 3 THEN
      ThisChar ← '7'
    ELSE
      ThisChar ← '*'
    ENDIF
  ENDIF
ENDIF
```

CASE options: merged, ordered, and replaced

- Two options that assign the same value merge into one with a value list:
1, 2 : ThisChar ← 'a'
- The options are tested **in order**. With 1 **TO** 50 : followed by 40 **TO** 60 :, a value of 45 takes the first match, so an assignment in a later option may **never be performed**.
- When the earlier options already cover every possible value, **OTHERWISE** is never reached either.
- Nested IFs that test several Booleans are clearer as **one condition per outcome**:
IF A **AND** B **AND** C **THEN CALL** Sub1()
IF A **AND** B **AND NOT** C **THEN CALL** Sub2()
■ **IF** A **THEN** is accepted in place of **IF** A = **TRUE THEN**.

When CASE cannot be used, and refining in five steps

CASE tests ONE variable against values or ranges. Use IF when the test:

compares **two variables** (IF Score > Best);
is a **compound** condition (Age > 12 AND Member = TRUE);
must **guard** another test (IF Count <> 0 before dividing).

Overlapping ranges: with 0 TO 50 first, the values 40–50 never reach 40 TO 60. That option is partly **unreachable**.

“Outline, using stepwise refinement, five steps.” Input 10 numbers; output the mean of the positive ones.

- 1 Set Total and Count to 0.
- 2 Repeat 10 times: input a number.
- 3 If it is greater than 0, add it to Total and add 1 to Count.
- 4 After the loop, if Count > 0, set the mean to Total / Count.
- 5 Output the mean, or a message if there were no positive numbers.

6

Sheet 11.2.2 – FOR loops: totals and counts

Sheet 11.2.2 – what you need to know

You must be able to:

- say how many times a FOR loop runs, with STEP
- write totals, counts, means, the largest and the smallest
- predict the output of nested loops

What the exam asks

“State the type of loop structure that would be the most appropriate and justify why.” [2]

Practise on exercise sheet 11.2.2, then try the real exam questions on past-paper sheet 11.2.

Count-controlled and pre-condition loops

Iteration 迭代 repeats a block. Three loops differ in how many times the body runs.

count-controlled loop 计数循环: use it when you **know how many times**

```
FOR i ← 1 TO 10  
    OUTPUT i  
NEXT i
```

A **STEP** changes the count:

FOR i ← 10 **TO** 1 **STEP** -1. Best for a fixed number of repeats, or for each element of an **array** 数组.

pre-condition loop 前测循环: tests **before** each pass

```
WHILE Total < 100 DO  
    INPUT N  
    Total ← Total + N  
ENDWHILE
```

The test comes first, so the body may run **zero** times.

Loop patterns: a total, a count, the largest

```
Total ← 0
BelowZero ← 0
FOR Day ← 1 TO 30
    INPUT Temp
    Total ← Total + Temp
    IF Temp < 0 THEN
        BelowZero ← BelowZero + 1
    ENDIF
NEXT Day
OUTPUT BelowZero, Total / 30
```

- **total**: set to 0 **before** the loop, add **inside** it
- **count**: set to 0, add 1 inside an IF
- **output after** the loop, not inside it
- **largest**: start with the **first value**, not with 0; replace it when a value is greater
- FOR Count ← 1 TO 5 runs 5 times; 0 TO 5 runs 6 times

Nested loops

```
FOR Row ← 1 TO 3
  Line ← ""
  FOR Star ← 1 TO Row
    Line ← Line & "*"
  NEXT Star
  OUTPUT Line
NEXT Row
```

- the **inner** loop runs completely for **each** pass of the outer loop
- the two loops need **different** counters
- NEXT Star comes before NEXT Row
- the output is *, **, ***: the inner statement runs $1 + 2 + 3 = 6$ times

7

Sheet 11.2.3 – WHILE and REPEAT

Sheet 11.2.3 – what you need to know

You must be able to:

- choose WHILE, REPEAT or FOR, and justify it
- write validation and rogue-value loops
- turn structured English into pseudocode

What the exam asks

“Write a pseudocode algorithm from this structured English description.” **[8]**

“State a more appropriate loop structure and justify your choice.” **[2]**

“Explain why this is not efficient and suggest a more efficient solution.” **[4]**

Practise on exercise sheet 11.2.3, then try the real exam questions on past-paper sheet 11.2.3.

The post-condition loop, and choosing the right loop

post-condition loop 后测循环: tests **after** each pass, so it always runs **at least once**

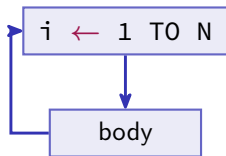
```
REPEAT  
  INPUT password  
UNTIL password = correctPassword
```

Justify the choice by whether the count is **known** and whether the body must run **at least once**.

If ...	use
the count is known	FOR
it may run zero times	WHILE
it must run at least once	REPEAT

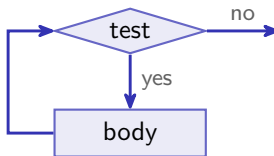
Where each loop tests its condition

FOR



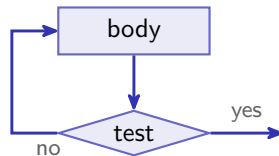
a set number
of passes

WHILE



test **before** the body:
may run 0 times

REPEAT



test **after** the body:
runs at least once

The three loops differ only in **where the condition is tested**: before the body, after it, or a set number of times.

Two loops you must know: validation and a rogue value

Validation: ask until the value is valid

```
REPEAT
  OUTPUT "Enter a mark, 0 to 100"
  INPUT Mark
UNTIL Mark >= 0 AND Mark <= 100
```

The mark must be input **at least once** before it can be tested: post-condition.

Rogue value: stop at -1 , and do not use it

```
Total ← 0
INPUT Price
WHILE Price <> -1
  Total ← Total + Price
  INPUT Price
ENDWHILE
```

The first input could be -1 , so the loop may run **zero** times: pre-condition.

Justifying the loop: the two marks

One mark for the **name** of the loop and one for the **reason**, in the scheme's words:

count-controlled

because the number of iterations is known before the loop starts

post-condition

because the loop body must be executed at least once

pre-condition

because the loop may not need to execute at all

A loop over the four elements of an array written as a **WHILE** with a counter is “not the most appropriate”: the count, four, is known, so a **FOR** loop fits.

Worked example: which loop suits each task?

Ask **how many times** the body runs, and **when the test happens**.

(a) print the 12 times table

FOR

the count, 12, is known in advance

(b) read numbers until the user enters 0

WHILE

the first input might already be 0, so test before the body: zero or more times

(c) ask for a password until it is correct

REPEAT ... UNTIL

you must ask at least once before there is anything to test: one or more times

The deciding question: must the body run at least once? **WHILE** may run zero times; **REPEAT** always runs once.

8

Sheet 11.2.4 – Trace tables

Sheet 11.2.4 – what you need to know

You must be able to:

- dry run an algorithm in a trace table
- find and correct a logic error with a trace
- name the loop type and its end condition

What the exam asks

“Complete the trace table by dry running the procedure Encode().”

[6]

“Identify the type of loop and state the condition that ends the loop.”

“Describe integration testing.”

[2]

Practise on exercise sheet 11.2.4, then try the real exam questions on past-paper sheet 11.2.4.

Dry running with a trace table

A **trace table** 跟踪表 records each variable's value as you **dry run** 手工跟踪 an algorithm, working through it by hand. It is a six-mark question on most Paper 2s.

```
DECLARE Count, Total : INTEGER
Count ← 1
Total ← 0
WHILE Total < 10 DO
    Total ← Total + Count * 2
    Count ← Count + 1
ENDWHILE
OUTPUT Count, Total
```

Count	Total	Total < 10	OUTPUT
1	0	TRUE	
2	2	TRUE	
3	6	TRUE	
4	12	FALSE	4, 12

Trace table rules that earn the marks

- 1 one column per variable, in the order the question gives
- 2 write a value only when it **changes**
- 3 start a new row each time the loop repeats
- 4 evaluate the condition with the current values, and stop the moment it is FALSE
- 5 put the output in its own column, exactly as it would appear
- 6 trace the algorithm **as written**: if it never stops, say so

Rewriting every variable on every row, or changing a value before the statement that changes it has run, both lose marks.

Finding a logic error with a trace table

Should output the total of 1 to 4, which is 10.

```
Total ← 0
FOR Index ← 1 TO 4
    Total ← Index
NEXT Index
OUTPUT Total
```

Index	Total	OUTPUT
	0	
1	1	
2	2	
3	3	
4	4	4

On the second pass Total should be 3, but it is 2: it is **replaced**, not added to. Answer with the **line, what it does wrong**, and the **corrected line**: Total ← Total + Index.

9

Sheet 11.3 – Procedures

Sheet 11.3 – what you need to know

You must be able to:

- write a procedure with parameters
- write its header and call it
- simplify a condition

What the exam asks

“Write pseudocode for procedure RandList().” [6]

“Write a simplified version of the conditional clause.” [1]

“Write pseudocode for the new module header.” [2]

Practise on exercise sheet 11.3, then try the real exam questions on past-paper sheet 11.3.

Procedures

Structured programming 结构化编程 builds a program from small named **subroutines** 子程序, each with one job.

```
PROCEDURE Greet(Name : STRING)
    OUTPUT "Hello, ", Name
ENDPROCEDURE

CALL Greet("Ada")
```

A **procedure** 过程 is a named block that **does an action**. It may take **parameters** 参数, but it does **not return a value**.

CALL Greet("Ada")



prints Hello, Ada

returns no value

When to use a subroutine

Use a subroutine when:

- the same logic appears in **more than one place**: write it once, call it many times
- a block has a **clear named purpose**: the name documents what it does
- the program is **complex**: break it into parts (**decomposition** 分解)
- you want to **test a piece in isolation**

Don't make them so tiny that the call costs more than the work inside.

10

Sheet 11.3.1 – Functions;
procedure or function?

Sheet 11.3.1 – what you need to know

You must be able to:

- write a function that returns a value
- choose a procedure or a function, and use the terms
- choose invalid test data for a function

What the exam asks

“Write pseudocode for function `Replace()`.” [6]

“Give four invalid strings that test distinct aspects of the required validation.” [4]

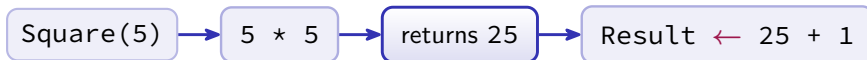
“Describe the function interface and two advantages of this modular approach.” [4]

Practise on exercise sheet 11.3.1, then try the real exam questions on past-paper sheet 11.3.1.

Functions

A **function** 函数 returns a value, which becomes part of an expression:

```
FUNCTION Square(x : INTEGER) RETURNS INTEGER  
    RETURN x * x  
ENDFUNCTION  
Result ← Square(5) + 1
```



Use a **procedure** when the subroutine performs an action; use a **function** when it computes a value for the caller.

Where each is appropriate

The syllabus asks *where in the construction of an algorithm* each one fits.

A procedure

where the **same group of steps** is needed at several points: validate an input, print a menu, swap two values. The steps are written once and **CALL**ed by name.

A function

where a **single value** must be calculated and then used in an expression, such as a total, a TRUE/FALSE result or the larger of two numbers.

The **return value** 返回值 replaces the call, which is why a function can sit inside a condition: `IF IsValid(Code) THEN`

Terminology

Term	Meaning
definition	the PROCEDURE ... ENDPROCEDURE (or function) block
call	where it is invoked
argument	a value passed in
parameter	the variable that receives it
return value	what a function passes back
procedure / function header	the first line, giving the name and parameters: PROCEDURE Name(params) or FUNCTION Name(params) RETURNS type
interface, or signature 签名	name + parameters + return type: what a caller must know to use it

Worked example: describe the header

Describe each term used in this header:

```
FUNCTION Pass2(Count : INTEGER) RETURNS BOOLEAN
```

Term	Meaning
FUNCTION	a subroutine that returns a value
Pass2	the identifier used to call it
Count	the parameter: the identifier that receives the argument passed in
INTEGER	the data type of the parameter
RETURNS BOOLEAN	the data type of the value the function returns

In **PROCEDURE** MyProc(Count : **INTEGER**, Message : **STRING**) both identifiers are **parameters**, used inside like local variables.

Converting a procedure into a function

- 1 change **PROCEDURE** to **FUNCTION** and add **RETURNS** with the type
- 2 replace the **OUTPUT**, or the **BYREF** parameter that carried the result out, with a **RETURN** statement
- 3 change every call so that the returned value is used:

Result ← Unpack(Text) instead of **CALL** Unpack(Text, Result)

For a “write the header” question, write the **whole line**: **FUNCTION**
Calculate(Expression : STRING) RETURNS INTEGER.

Test data for one function, and exposing a fault

IsValid(Password): 8–12 characters, at least one digit, at least one capital

test data	tests	expected
"ABCDEFGG1"	boundary: 8 long	TRUE
"Ab1"	too short	FALSE
"ABCDEFGHIJK12"	too long (13)	FALSE
"ABCDEFGH1"	no capital	FALSE
"ABCDEFGHIJ"	no digit	FALSE

Each invalid string breaks **one** rule only, so a FALSE shows which check works.

“Explain two ways of exposing the fault.” [2]

Dry run the module with a trace table, using data that reaches the faulty line.

Run it in the IDE with a **breakpoint** and **watch** the variables, or add OUTPUT statements to show their values.

11

Sheet 11.3.2 – BYVAL, BYREF and scope

Sheet 11.3.2 – what you need to know

You must be able to:

- trace calls that pass BYVAL and BYREF
- explain local and global scope
- expose a fault in a module

What the exam asks

“Complete the following table for each of the two methods.” [6]

“Explain why it does not work as expected.” [2]

“Explain two ways of exposing the fault.” [2]

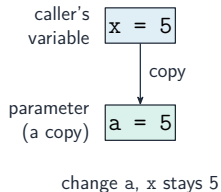
Practise on exercise sheet 11.3.2, then try the real exam questions on past-paper sheet 11.3.2.

Parameters: by value and by reference

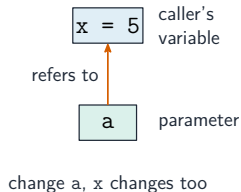
A **parameter** is a variable a subroutine declares to receive input; the values the caller supplies are **arguments** 实参.

- **pass by value** 传值: the routine gets a **copy**; changes inside it do not affect the caller. Use it for inputs the routine only reads.
- **pass by reference** 传引用: the routine gets a **reference** to the caller's variable; changes **do** affect the caller. Use it when the routine must update a parameter.

pass by value



pass by reference



Value copies; reference lets the routine change the caller's own variable

BYVAL and BYREF in the header

```
PROCEDURE Swap(BYREF a : INTEGER,  
               BYREF b : INTEGER)  
  DECLARE Temp : INTEGER  
  Temp ← a  
  a ← b  
  b ← Temp  
ENDPROCEDURE
```

Cambridge pseudocode writes the mode, BYVAL or BYREF, in the header **before each parameter**.

If neither is written, BYVAL is assumed. A routine that must change the caller's variable, such as Swap or a procedure that updates a running total, needs BYREF.

Worked example: what is output?

```
PROCEDURE Adjust(BYREF X : INTEGER,  
                 BYVAL Y : INTEGER)
```

```
    X ← X + Y
```

```
    Y ← Y * 2
```

```
ENDPROCEDURE
```

```
A ← 5
```

```
B ← 3
```

```
CALL Adjust(A, B)
```

```
OUTPUT A, B
```

X is a **reference** to A, so A becomes 8

Y is a **copy** of B, so doubling Y leaves B at 3

the output is 8, 3

Had the header said **BYVAL** X, A would still be 5.

Local and global variables

A **local variable** 局部变量 is declared inside a subroutine and exists only while it runs. A **global variable** 全局变量 is declared outside and is visible everywhere.

The region where a name is visible is its **scope** 作用域.

Prefer locals and parameters: heavy use of globals makes code hard to follow and test.

global scope: the whole program

Total

global: visible everywhere

local scope: procedure Calc

Temp

local: exists only
while Calc runs

Why local variables: the exam answer

The one-line difference: a global variable can be accessed from **anywhere** in the program, a local variable only **inside the subroutine that declares it**. Benefits of local variables:

1 the same identifier can be used in another subroutine **without a clash**

2 the value **cannot be changed accidentally** by other parts of the program

3 the memory is **released** when the subroutine ends

4 the subroutine is **self-contained**: tested on its own and reused in another program

A local variable is new on every call

MyOutput() adds one letter to MyString each time it is called:

✓ MyString is global

call 1: MyString = "A"

call 2: MyString = "AB"

call 3: MyString = "ABC"

one variable: the text builds up

✗ MyString is local

call 1: MyString = "A"

call 2: MyString = "B"

call 3: MyString = "C"

a new, empty variable each call

- a local variable is created **each time** the subroutine is called and destroyed when it returns, so it cannot carry a value from one call to the next
- a string built up over repeated calls must therefore be **global** (or passed BYREF); made local, the procedure “does not work as expected”

12

Sheet 11.3.3 – Efficient pseudocode and modules

Sheet 11.3.3 – what you need to know

You must be able to:

- write a whole module, in six parts
- make pseudocode efficient and easy to read
- plan tests for a function

What the exam asks

“Write pseudocode for an algorithm to implement your own version of the MID() function.”

[4]

“Complete the test plan for five tests that could be performed on function Status().”

“Complete the table by writing the text that should replace each label A to F.”

Practise on exercise sheet 11.3.3, then try the real exam questions on past-paper sheet 11.3.3.

The six parts of a module answer

Half of Paper 2 is “write pseudocode for module X”. The scheme awards a mark **per feature**, so a module that is not finished still scores for every correct part.

1 The header

as the question gives it, with BYREF where an argument must change

2 Local declarations

DECLARE every local with its type; set counters and totals to 0

3 The loop

FOR over a given size; WHILE NOT EOF(. . .) for a file

4 The condition

the right comparison and the right boundary, on the right item

5 The update

inside the branch: the count increased, the value stored, or the message output

6 The end

RETURN once, after the loop; every IF, FOR and WHILE closed

Worked example: IsValid, the plan

Question

Write a function IsValid(Code : STRING) that returns TRUE when Code is two capital letters followed by four digits (the **format** 格式 AB1234), and FALSE otherwise.

- check the **length first**, so MID never reads past the end
- visit positions 1 to 6: capitals at 1 and 2, digits at 3 to 6
- RETURN FALSE at the first bad character, RETURN TRUE after the loop

```
FUNCTION IsValid(BYVAL Code : STRING)
    RETURNS BOOLEAN
    DECLARE Index : INTEGER
    DECLARE Ch : STRING
    IF LENGTH(Code) <> 6 THEN
        RETURN FALSE
    ENDIF
    // ... the character loop, next
```

Worked example: IsValid, each character

```
FOR Index ← 1 TO 6
  Ch ← MID(Code, Index, 1)
  IF Index <= 2 THEN
    IF Ch < "A" OR Ch > "Z" THEN
      RETURN FALSE
    ENDIF
  ELSE
    IF Ch < "0" OR Ch > "9" THEN
      RETURN FALSE
    ENDIF
  ENDIF
NEXT Index
RETURN TRUE
ENDFUNCTION
```

Validation 验证 like this
returns a BOOLEAN, so the
caller decides what to do:

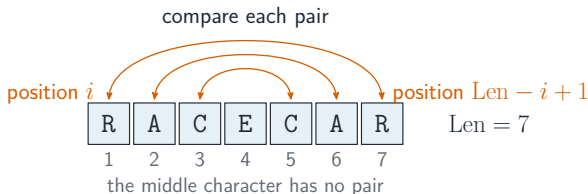
```
IF IsValid(Entry) THEN
  CALL Save(Entry)
ELSE
  OUTPUT "Invalid"
ENDIF
```

The **caller** outputs the
message: a function
calculates, a procedure acts.

Worked example: IsPalindrome, the method

Question

Write a function `IsPalindrome(Word : STRING)` that returns `TRUE` when `Word` reads the same backwards, such as "RACECAR".



Compare the characters from the two ends, moving inwards: position `Index` pairs with position `Len - Index + 1`, and only the first half needs testing.

Worked example: IsPalindrome, the function

```
FUNCTION IsPalindrome(BYVAL Word : STRING) RETURNS BOOLEAN
  DECLARE Len, Index : INTEGER
  Len ← LENGTH(Word)
  FOR Index ← 1 TO Len DIV 2
    IF MID(Word, Index, 1) <> MID(Word, Len - Index + 1, 1) THEN
      RETURN FALSE
    ENDIF
  NEXT Index
  RETURN TRUE
ENDFUNCTION
```

Len DIV 2 stops at the middle: an odd-length word's middle character has no pair, and needs no test.

Three tools that answer most string modules

Most string modules need a **FOR** over the positions, MID(s, i, 1) and &:

Task	The idea
count a character	IF MID(s, i, 1) = Ch THEN Count ← Count + 1
replace every instance	add NewChar or the original character to NewString
hide a card number	add '*' up to position Len - 4, then the last four digits
write your own MID()	join the characters from Start to Start + Length - 1

MID past the end of the string is a **run-time error**: check LENGTH first.

Pseudocode that is easy to understand

Three features, the answer to a “state three features” question:

Meaningful identifiers

Total, not t

Indentation

of the statements
inside each construct

Comments

// ... that ex-
plain the purpose

Also accepted: keywords in capitals, one statement per line, and blank lines between sections.

Writing efficient pseudocode

- **move invariants out of loops:** if a value (an **invariant** 不变量) does not change with the loop counter, compute it once before the loop
- **exit a loop early** when the answer is found: stop a **linear search** 线性查找 as soon as the target appears
- **avoid redundant work:** store a result and reuse it instead of recomputing it
- **choose the right data structure:** an array beats many separate variables when the items belong together
- **replace deep nested IFs with CASE** when testing one value against many
- **comment the intent**, not the mechanics: `// validate the postcode`, not `// loop 6 times`
- **use meaningful names** (`numberOfPupils`, not `n`) and **initialise variables** before use

Move work that never changes out of the loop

Slower: the same result, recomputed on every pass

```
FOR i ← 1 TO n  
  x ← Slow()  
  CALL Use(x, i)  
NEXT i
```

Faster: computed once, before the loop

```
x ← Slow()  
FOR i ← 1 TO n  
  CALL Use(x, i)  
NEXT i
```

Anything whose answer does not change between passes belongs **outside** the loop: compute it once, store it, and use the stored value.

13

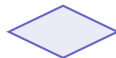
Beyond the sheets

From design to code

Turn a design, a **flowchart** 流程图 (a **program flowchart**) or **structured English** 结构化英语, into **pseudocode** 伪代码 and then into a real language:

- 1 find the **variables** 变量 and their **data types** 数据类型
- 2 input/output boxes become **INPUT** / **OUTPUT**
- 3 diamonds become **IF ... ENDIF** or **CASE**
- 4 loop arrows become **WHILE**, **REPEAT** or **FOR**
- 5 process boxes become assignments
- 6 trace a small input to check

flowchart



pseudocode

INPUT / **OUTPUT****IF ... THEN** or **CASE** $x \leftarrow \text{expression}$ **WHILE** / **FOR** / **REPEAT**

Worked example: which constructs does each line use?

Pseudocode	Selection	Iteration	Subroutine
IF Ready = TRUE THEN CALL Start() ENDIF	✓		✓
FOR I ← 1 TO 20 ... NEXT I		✓	
WHILE NOT IsFull() ... ENDWHILE		✓	✓
CASE OF Key ... OTHERWISE ... ENDCASE	✓		

IF and **CASE** are selection; **FOR**, **WHILE** and **REPEAT** are iteration; a name followed by brackets, such as `Start()` or `IsFull()`, is a call to a procedure or function, wherever it appears, including inside a condition.

Interfaces, and arrays as parameters

A new module's interface

When a program gains a new module, the interface is agreed **first**:

- its name
- its parameters: how many, in what order, of what type
- its return type
- any global data it reads or writes

A module that sends a reminder before a due date needs the record (or its index) as a parameter and returns nothing, so it is a **procedure**; the main program calls it once per record.

An **array** parameter is passed **by reference**, so a procedure that writes into an array changes the caller's array.

Worked example: CountAbove

Question

A global array `Score : ARRAY[1:50] OF INTEGER` holds test scores. Write a function `CountAbove(Limit : INTEGER)` that returns how many scores are greater than `Limit`.

Where the marks are

- 1 the header and **RETURNS** `INTEGER`
- 2 `Count` declared and set to 0
- 3 a loop over `all 50` elements
- 4 the comparison `> Limit`, not `>=`
- 5 the count updated `inside` the **IF**
- 6 **RETURN** `Count` `after` the loop

The main program uses the value in an expression or an output:

OUTPUT `"Above 70: ",`
`CountAbove(70)`

CountAbove: the solution, part by part

```
FUNCTION CountAbove(BYVAL Limit : INTEGER) // 1
    RETURNS INTEGER
    DECLARE Index, Count : INTEGER // 2
    Count ← 0
    FOR Index ← 1 TO 50 // 3
        IF Score[Index] > Limit THEN // 4
            Count ← Count + 1 // 5
        ENDIF
    NEXT Index
    RETURN Count // 6
ENDFUNCTION
```

Files (1 of 2): writing the data out

Values in variables disappear when the program ends, so a module that must keep data for the next run writes it to a file.

```
OPENFILE "scores.txt" FOR WRITE
FOR Index ← 1 TO 50
    WRITEFILE "scores.txt", NUM_TO_STR(Score[Index])
NEXT Index
CLOSEFILE "scores.txt"
```

Open **once**, before the loop; one WRITEFILE per line **inside** the loop; close **once**, after it.

Files (2 of 2): reading it back

```
OPENFILE "scores.txt" FOR READ
WHILE NOT EOF("scores.txt") DO
    READFILE "scores.txt", Line
    OUTPUT Line
ENDWHILE
CLOSEFILE "scores.txt"
```

The marks

- the file opened in the **right mode**
- the read or write **inside** the loop
- the file closed **once**, after the loop

Topic 10 has the full file section.

Testing and errors

Three kinds of error, each found in a different way:

Error	What it is	Example	Found by
syntax error 语法错误	a statement that breaks the rules of the language	a missing ENDIF ; OUTPT "Hi"	the translator, before the program runs
run-time error 运行时错误	the program runs, but a statement cannot be carried out	division by zero; an array index of 0 or 51; an invalid parameter; a loop that never ends	running it: the program stops or hangs
logic error	the program runs to the end, but the output is wrong	> where >= was needed; a total never set to 0	a trace table and chosen test data

Finding errors with an IDE

An **IDE** 集成开发环境 helps find run-time and logic errors:

- a **breakpoint** 断点 stops the program at a chosen line
- **single stepping** 单步执行 then runs one statement at a time
- the report or watch window shows each variable's value at that moment, so the line where a value goes wrong is seen directly

Test methods and test data are in topic 12.



Write the code, and test it as you go

14

Recap

Definitions the examiner accepts (1 of 3)

A definition question is marked against fixed wording. Learn these exactly.

Term	Definition
procedure	a subroutine that carries out a task (a sequence of steps) and does not return a value; it is called with CALL
function	a subroutine that returns a single value to the point where it was called, so it can be used in an expression
parameter	the identifier in a subroutine header that receives a value or a reference when the subroutine is called
argument	the value (or variable) supplied in the call, matched to a parameter
passing by value	a copy of the argument's value is given to the subroutine, so changes inside it do not affect the original variable
passing by reference	the address of the variable is given to the subroutine, so changes inside it change the original variable

Definitions the examiner accepts (2 of 3)

Term	Definition
header	the first line of a subroutine definition: its name, its parameters and, for a function, its return type
interface	what a calling program must know to use a subroutine: its name, its parameters (number, order, type) and its return type
return value	the value a function passes back to the expression that called it
local variable	declared inside a subroutine; it exists only while the subroutine runs and can be used only inside it
global variable	declared outside every subroutine; it can be used anywhere in the program
count-controlled loop	repeats a fixed number of times, controlled by a counter (FOR ... NEXT)

Definitions the examiner accepts (3 of 3)

Term	Definition
pre-condition loop	tests its condition before each iteration, so the body may never run (WHILE ... ENDWHILE)
post-condition loop	tests its condition after each iteration, so the body runs at least once (REPEAT ... UNTIL)
constant	a named value that cannot change while the program runs
subroutine	a self-contained block of code that performs a task and is called by name: a procedure or a function
library routine	a subroutine that has already been written and tested, and is available to be called from a program

Exam tips

- Distinguish a **procedure** (no return value) from a **function** (returns a value); know **pass by value vs by reference**.
- Choose the right loop: **count-controlled (FOR)** when the number of repeats is known, **condition-controlled (WHILE / REPEAT)** otherwise.
- Distinguish **local vs global** variables and scope; prefer local variables in reusable modules.
- Use the insert's **exact routine names** and parameter order; UCASE and VAL are IGCSE names and score nothing here.
- In a “write pseudocode” answer the **header, declarations, loop, condition, update and RETURN** each carry a mark: write all six parts, even if one is uncertain.

Common mistakes (1 of 2)

The mistake

calling a function and not using what it returns

passing a length one out: 6 for a seven-element array

closing a file inside the loop that reads it
using the input as a filename directly

leaving constructs open

What earns the mark

assign it, or use it in an expression or output:
`Sorted ← BubbleSort(MyArray, 7)`

decide whether the parameter is a **length** or an **index**, and check the last element is visited

open once, and close once, after the loop

add the extension the question gave:
`FileName ← Choice & ".txt"`

every **IF** needs its **ENDIF**, every **FOR** its **NEXT**, every **WHILE** its **ENDWHILE**, every function its **RETURN**

Common mistakes (2 of 2)

The mistake

wrong boundaries: `>` for “at least”

a counter or total never set to 0

in a trace table, rewriting every variable on every row

half a condition: **IF** `x = 3` **OR** `4`

a value that must survive between calls, declared as local

What earns the mark

“at least” is `>=`; and a **FOR** over an array declared `[1:50]` starts at 1, not 0

set it to 0 before the loop

write a value only when it changes, and only after the statement that changes it

each side of **OR** and **AND** a complete comparison; and `&` joins strings, not `+`

a running total or a string built over several calls is global, or **BYREF**

Topic 11 – key takeaways

1

Basics

declare with a type; assign with an arrow; the insert's exact names

2

Selection and loops

FOR when the count is known;
WHILE may run zero times;
REPEAT at least once

3

Subroutines

a procedure acts, a function returns; BYVAL copies, BYREF changes the caller

4

Module answers

header, declarations, loop, condition, update, RETURN: six marks

Check yourself

- 1 Which loop may run zero times?
- 2 What does `MID("Program", 4, 2)` return?
- 3 Which mode lets a procedure change the caller's variable?
- 4 What is $7 \text{ DIV } 2 + 7 \text{ MOD } 2$?



Answers 1. WHILE 2. "gr" 3. BYREF 4. 4