AS-Level Pseudocode Reference

exercises.lol exercises.com.cn

Contents

1 Writing pseudocode **3**
1.1 Keywords, identifiers and case . . . 3
1.2 Indentation and comments . . . 4

2 Data types, variables and constants **6**
2.1 The six data types . . . 6
2.2 DECLARE and assignment . . . 7
2.3 CONSTANT . . . 8

3 Input, output and operators **10**
3.1 INPUT and OUTPUT . . . 10
3.2 Arithmetic, DIV and MOD . . . 11
3.3 Relational and logic operators . . . 12

4 Selection **14**
4.1 IF, THEN, ELSE, ENDIF . . . 14
4.2 Nested IF . . . 15
4.3 CASE OF . . . 16

5 Iteration **20**
5.1 FOR loops and STEP . . . 20
5.2 WHILE, the pre-condition loop . . . 21
5.3 REPEAT, the post-condition loop . . . 22
5.4 Choosing the right loop . . . 23

6 Arrays **25**
6.1 One-dimensional arrays . . . 25
6.2 Two-dimensional arrays . . . 26
6.3 Totals, counts, largest and smallest . . . 28

7 Strings and built-in functions **31**
7.1 Joining strings with & . . . 31
7.2 LEFT, RIGHT, MID and LENGTH . . . 32
7.3 Case and character codes . . . 33
7.4 Numbers and strings . . . 35

8 Procedures and functions **37**
8.1 PROCEDURE and CALL . . . 37
8.2 FUNCTION and RETURN . . . 38
8.3 BYVAL and BYREF . . . 39
8.4 Local and global variables . . . 41

9 Records and user-defined types **44**
9.1 Defining a record type . . . 44
9.2 Arrays of records . . . 46
9.3 Other user-defined types . . . 47

10 Text files **49**
10.1 Opening and closing a file . . . 49
10.2 Writing and appending . . . 50
10.3 Reading to the end of a file . . . 51

11 Standard algorithms **54**
11.1 Linear search . . . 54
11.2 Bubble sort . . . 55
11.3 Validating input . . . 58

12 Stacks, queues and linked lists **62**
12.1 A stack in an array . . . 62
12.2 A circular queue in an array . . . 64
12.3 A linked list in arrays . . . 66

13 Quick reference **69**
13.1 Every keyword on one page . . . 69
13.2 Coming from IGCSE 0478 . . . 71
13.3 Writing a Paper 2 answer . . . 72

exercises.lol exercises.com.cn

1 Writing pseudocode

1.1 ■ Keywords, identifiers and case

Pseudocode is a way to write an algorithm that looks like a program but is not tied to one programming language. In the 9618 exam, every algorithm in a question is printed in pseudocode, and many answers must be written in it. Cambridge publishes one guide that sets the rules, and this reference follows it.

You do not need perfect pseudocode to get the marks. You do need pseudocode that a marker can read **without guessing**. The rules below make that easy.

1.1.1 Keywords are in capitals

A **keyword** is a word with a fixed meaning, such as `IF`, `WHILE`, `DECLARE` or `OUTPUT`. Keywords are always written in **upper case** upper case .

```
DECLARE Total : INTEGER
Total ← 10 + 5
OUTPUT Total
```

Here `DECLARE`, `INTEGER` and `OUTPUT` are keywords. `Total` is a name that we chose.

1.1.2 Identifiers

An **identifier** is the name you give to a variable, a constant, a procedure or a function. The 9618 rules are:

- It **starts with a letter** (A–Z or a–z), never a digit.
- After the first letter it may contain letters, digits (0–9) and the underscore `_`.
- No spaces, no accented letters, and no other symbols.
- It must **not** be a keyword. `Output` or `Case` cannot be a variable name.

Identifier	Allowed?	Why
`TotalScore`	yes	letters only
`Player2`	yes	a digit after the first letter is fine
`max_value`	yes	9618 allows the underscore and a small first letter
`2ndPlace`	**no**	starts with a digit
`Total Score`	**no**	contains a space
`WHILE`	**no**	it is a keyword

Cambridge writes identifiers in **mixed case** mixed case : each new word starts with a capital, as in `NumberOfPlayers`. This is the style you will see in every paper.

1.1.3 Choose names that explain themselves

Good names make your answer easier to mark. `Average` says what the value means; `A` does not. Single letters are fine only where they are a normal habit: `i` and `j` for array indices, `X` and `Y` for coordinates.

1.1.4 Identifiers are case-insensitive

`Count`, `COUNT` and `count` are treated as the **same** name. So never use two names that differ only in case, such as `CountDown` and `Countdown`. A marker will see one variable, and your algorithm will not mean what you wanted.

```
DECLARE Count : INTEGER
Count ← 3
count ← count + 1
OUTPUT COUNT
```

This prints `4`: all three spellings reach one variable. It runs, but it is bad style. Pick one spelling and keep it.

Common mistakes

- Writing keywords in small letters, such as `if` or `output`. Write `IF` and `OUTPUT`.
- Using a keyword as a name, for example `DECLARE String : STRING`.
- Putting a space inside a name: write `FirstName`, not `First Name`.

1.2 ■ Indentation and comments

1.2.1 Indentation shows what is inside what

Indentation means moving a line to the right. In pseudocode it shows that a line belongs **inside** the statement above it, for example inside a loop or an `IF`. The 9618 guide usually indents by **three spaces**.

```
DECLARE Counter : INTEGER
FOR Counter ← 1 TO 3
   OUTPUT "Line ", Counter
   IF Counter = 3 THEN
      OUTPUT "Done"
   ENDIF
NEXT Counter
```

- `OUTPUT "Line "` and the `IF` are inside the `FOR` loop, so they move in 3 spaces.
- `OUTPUT "Done"` is inside the `IF`, so it moves in 3 more.
- `ENDIF` lines up with its `IF`; `NEXT` lines up with its `FOR`.

The program would run the same with no indentation at all. But a marker reads the layout to see where each block starts and ends, so **always indent**. An answer with no indentation is hard to follow and may lose marks when the structure is unclear.

1.2.2 Comments

A **comment** is a note for the human reader. The computer ignores it. A comment starts with two forward slashes `//` and runs to the end of the line.

```
// Work out the area of a rectangle
DECLARE Width : INTEGER
DECLARE Height : INTEGER
Width ← 4
Height ← 7
OUTPUT Width * Height   // area in square metres
```

- Usually a comment goes on its **own line**, before the code it explains, at the same indentation.
- A short comment may go at the end of one line, as in the last line above.
- A comment over several lines needs `//` at the start of **each** line.

1.2.3 Line numbers

Some questions print **line numbers** line numbers on the left, so that a question can say "line 07 has an error". The numbers are **not** part of the pseudocode. When you correct a line, write the line number and the new code, for example:

```
Line 07:  WHILE Index <= 10
```

1.2.4 Practice

Find what is wrong with each line. Then open the answer.

1. `DECLARE 1stName : STRING`

2. `declare Total : INTEGER`

3. `DECLARE Average Mark : REAL`

<details> <summary>Model answer</summary>

1. The identifier starts with a digit. Use `FirstName`.
2. The keyword must be in capitals: `DECLARE`.
3. An identifier cannot contain a space. Use `AverageMark`.

</details>

Common mistakes

- Forgetting to indent the body of a loop or an `IF`.
- Ending a block at the wrong level: `ENDIF` must line up with its own `IF`.
- Writing a comment with one slash `/`. One slash means **divide**; a comment needs `//`.

exercises.lol exercises.com.cn

2 Data types, variables and constants

2.1 ■ The six data types

Every value has a **data type** . The type says what kind of value it is and what you can do with it. 9618 pseudocode has six basic types.

Type	What it holds	Example values
`INTEGER`	a whole number whole number	`5`, `-3`, `0`
`REAL`	a number that can have a fractional part fractional part	`4.7`, `0.3`, `-4.0`
`CHAR`	exactly one character	`'x'`, `'C'`, `'@'`
`STRING`	zero or more characters	`"Hello"`, `""`
`BOOLEAN`	`TRUE` or `FALSE`	`TRUE`
`DATE`	a calendar date	`02/01/2005`

2.1.1 How to write each value

A value written straight into the code is called a **literal** . Each type has its own way of writing one:

- **INTEGER**: normal digits, `25` or `-3`.
- **REAL**: always at least one digit **on each side** of the decimal point. Write `0.5`, not `.5`, and `4.0`, not `4.`.
- **CHAR**: one character in **single** quotes: `'A'`.
- **STRING**: characters in **double** quotes: `"A"`. The empty string empty string `""` has no characters at all.
- **BOOLEAN**: the words `TRUE` and `FALSE`, with no quotes.
- **DATE**: normally `dd/mm/yyyy`. Dates are written differently around the world, so it is good practice to say which format you use.

```
DECLARE Price : REAL
DECLARE Initial : CHAR
DECLARE Name : STRING
DECLARE Paid : BOOLEAN
Price ← 12.50
Initial ← 'M'
Name ← "Mei Lin"
Paid ← TRUE
OUTPUT Name, " (", Initial, ") paid ", Price, ": ", Paid
```


exercises.lol exercises.com.cn

2.1.2 Which type should I choose?

- Something you **count** (people, goals, items): `INTEGER`.
- Something you **measure** or that can have decimals (money, height, average): `REAL`.
- A yes/no answer, or a flag flag that remembers whether something happened: `BOOLEAN`.
- A phone number or an ID such as `"007"`: `STRING`. You never do arithmetic on it, and an `INTEGER` would lose the leading zeros.

Common mistakes

- `'Hello'` in single quotes. Single quotes hold **one** character only; a string needs double quotes.
- `"TRUE"` in quotes. That is a string of four letters, not a Boolean.
- Storing an average in an `INTEGER`. An average is usually not a whole number, so use `REAL`.

2.2 ■ DECLARE and assignment

2.2.1 DECLARE

A **variable** is a named place in memory that holds one value, and the value can change while the program runs. Before you use a variable, you **declare** it: you give its name and its data type.

```
DECLARE <identifier> : <data type>
```

```
DECLARE Counter : INTEGER
DECLARE TotalToPay : REAL
DECLARE GameOver : BOOLEAN
Counter ← 0
TotalToPay ← 0.0
GameOver ← FALSE
OUTPUT Counter, " ", TotalToPay, " ", GameOver
```

Declare variables **at the top** of your answer, one per line. The guide says declaring is good practice, and many mark schemes give a mark for it. If a question says ”declare all variables”, you must do it.

2.2.2 Assignment

Assignment puts a value into a variable. The operator is a **left arrow** ←.

```
<identifier> ← <value>
```

Read `Counter ← 0` as "Counter **becomes** 0" or "Counter is given 0". The right side is worked out first, then the result is stored on the left. That is why this line makes sense:

```
DECLARE Counter : INTEGER
Counter ← 5
Counter ← Counter + 1
OUTPUT Counter
```

The right side `Counter + 1` uses the **old** value, 5. The result, 6, then replaces it.

- The left side must be **one variable** (or one array element, or one field of a record).
- The right side can be any expression that gives a value of the same type.
- Typing on a keyboard? The site also accepts `<-` and turns it into `←`.

Common mistakes

- `Counter = 0` to assign. In pseudocode `=` **compares** two values; it never stores one.
- `0 ← Counter`, with the sides swapped. The variable that changes is always on the left.
- Using a variable before it has a value, for example adding to `Total` before `Total ← 0`. Always give a starting value. This is called **initialising** the variable.

2.3 ■ CONSTANT

A **constant** is a named value that **never changes** while the program runs.

```
CONSTANT <identifier> = <value>
```

```
CONSTANT HourlyRate = 6.50
CONSTANT DefaultText = "N/A"
DECLARE Hours : INTEGER
Hours ← 8
OUTPUT "Pay: ", Hours * HourlyRate
OUTPUT "Comment: ", DefaultText
```

Note the **equals sign** `=` here, not the arrow. A constant is defined once; it is not assigned.

2.3.1 Why use a constant?

- `HourlyRate` tells the reader what `6.50` means. A bare number in the middle of code does not.
- If the rate changes, you change **one line**, not every place the number appears.

exercises.lol exercises.com.cn

2.3.2 The rules

- Put constants at the **start** of the program.
- The value must be a **literal**, such as `6.50` or `"N/A"`. It cannot be a variable, another constant or a calculation. `CONSTANT Total = Price * 2` is not allowed.
- A program must never assign to a constant. `HourlyRate ← 7.00` is an error.

2.3.3 Practice

1. Write declarations for: a person's age, their height in metres, whether they are a member, and their first initial.

2. A shop has a fixed tax rate of 0.08. Write the constant, then output the tax on a price of 50.

<details> <summary>Model answer</summary>

```
DECLARE Age : INTEGER
DECLARE Height : REAL
DECLARE IsMember : BOOLEAN
DECLARE FirstInitial : CHAR
Age ← 17
Height ← 1.72
IsMember ← TRUE
FirstInitial ← 'J'
OUTPUT Age, " ", Height, " ", IsMember, " ", FirstInitial
```

```
CONSTANT TaxRate = 0.08
DECLARE Price : REAL
Price ← 50.0
OUTPUT "Tax: ", Price * TaxRate
```

The output of the second program is `Tax: 4.0`.

</details>

Common mistakes

- `CONSTANT TaxRate ← 0.08`. That is the **IGCSE** form. In 9618 a constant uses `=`.
- Writing `DECLARE` in front of a constant. A constant uses the keyword `CONSTANT` only.

3 Input, output and operators

3.1 ■ INPUT and OUTPUT

3.1.1 INPUT

`INPUT` reads a value from the user and stores it in a variable.

```
INPUT <identifier>
```

The identifier must be a variable. It can also be one element of an array or one field of a record.

```
DECLARE Age : INTEGER
OUTPUT "How old are you?"
INPUT Age
OUTPUT "Next year you will be ", Age + 1
```

When you press Run, the page asks you to type the input first.

- Show a **prompt** first: an `OUTPUT` that tells the user what to type. Mark schemes often give a mark for it.
- `INPUT` takes **one** variable. To read two values, write two `INPUT` lines.

3.1.2 OUTPUT

`OUTPUT` shows one or more values on the screen. Separate several values with **commas**.

```
OUTPUT <value(s)>
```

```
DECLARE Lives : INTEGER
DECLARE Score : INTEGER
Lives ← 3
Score ← 1250
OUTPUT Score
OUTPUT "You have ", Lives, " lives left"
```

The values are printed one after another on the same line, with nothing added between them. That is why `"You have "` ends with a space.

Common mistakes

- `INPUT "Enter age", Age`. `INPUT` does not show a message. Use an `OUTPUT` line first.
- `OUTPUT "Total is Total"`. The word inside the quotes is just text. Put the variable outside the quotes: `OUTPUT "Total is ", Total`.
- `PRINT` instead of `OUTPUT`. `PRINT` is Python, not Cambridge pseudocode.

3.2 ■ Arithmetic, DIV and MOD

3.2.1 The arithmetic operators

Operator	Meaning	Example	Result
`+`	add	`7 + 2`	`9`
`-`	subtract	`7 - 2`	`5`
`*`	multiply	`7 * 2`	`14`
`/`	divide	`7 / 2`	`3.5`
`DIV`	whole-number division	`7 DIV 2`	`3`
`MOD`	remainder after division	`7 MOD 2`	`1`

- `/` always gives a `REAL`, even when both numbers are integers: `10 / 2` is `5.0`.
- `*` and `/` are done before `+` and `-`, as in normal maths. Use **brackets** to make the order clear: `(A + B) / 2`.

```
DECLARE A : INTEGER
DECLARE B : INTEGER
A ← 10
B ← 4
OUTPUT A + B * 2
OUTPUT (A + B) * 2
OUTPUT A / B
```

This prints `18`, `28` and `2.5`.

3.2.2 DIV and MOD

`DIV` gives the **quotient** : how many whole times one number goes into another. `MOD` gives the **remainder** : what is left over. In 9618 both are written **between** their two numbers, like `+`.

```
DECLARE Minutes : INTEGER
Minutes ← 135
OUTPUT Minutes DIV 60, " hours and ", Minutes MOD 60, " minutes"
```

135 minutes is 2 hours ($2 \times 60 = 120$) with 15 minutes left over.

Three uses you will meet again and again:

- **Even or odd**: `Number MOD 2 = 0` is `TRUE` for an even number.
- **Last digit**: `Number MOD 10` gives the last digit of a positive whole number.
- **Remove the last digit**: `Number DIV 10`. So `472 DIV 10` is `47`.

exercises.lol exercises.com.cn

```
DECLARE Number : INTEGER
DECLARE DigitSum : INTEGER
Number ← 472
DigitSum ← 0
WHILE Number > 0
   DigitSum ← DigitSum + (Number MOD 10)
   Number ← Number DIV 10
ENDWHILE
OUTPUT DigitSum
```

This adds the digits $4 + 7 + 2$ and prints `13`.

Common mistakes

- `DIV(7, 2)` or `MOD(7, 2)`. That is the **IGCSE** form. In 9618 write `7 DIV 2`.
- Using `/` when a whole number is needed, for example for an array index. `7 / 2` is `3.5`, which is not a valid index; use `7 DIV 2`.

3.3 ■ Relational and logic operators

3.3.1 Relational operators

A **relational operator** compares two values. The answer is always a `BOOLEAN`: `TRUE` or `FALSE`.

Operator	Meaning
`=`	equal to
`<>`	not equal to
`<`	less than
`>`	greater than
`<=`	less than or equal to
`>=`	greater than or equal to

```
DECLARE Mark : INTEGER
Mark ← 65
OUTPUT Mark >= 50
OUTPUT Mark = 100
OUTPUT Mark <> 0
```

This prints `TRUE`, `FALSE` and `TRUE`.

3.3.2 Logic operators

The **logic operators** `AND`, `OR` and `NOT` join or change Boolean values.

Expression	It is TRUE when...
`A AND B`	**both** A and B are TRUE
`A OR B`	**at least one** of A and B is TRUE
`NOT A`	A is FALSE

A range test needs `AND` with **two** full comparisons:

```
DECLARE Age : INTEGER
DECLARE HasTicket : BOOLEAN
Age ← 15
HasTicket ← FALSE
OUTPUT Age >= 13 AND Age <= 19
OUTPUT Age < 12 OR HasTicket
OUTPUT NOT HasTicket
```

This prints `TRUE`, `FALSE` and `TRUE`.

When `AND` and `OR` are mixed, use **brackets** so the meaning is clear: `(Day = "Sat" OR Day = "Sun") AND NOT Raining`.

3.3.3 Practice

1. What does `29 DIV 4` give? What does `29 MOD 4` give?

2. Write one expression that is TRUE when `Temp` is **outside** the range 18 to 25.

<details> <summary>Model answer</summary>

1. `29 DIV 4` is `7` and `29 MOD 4` is `1`, because $4 \times 7 = 28$ and 1 is left over.
2. `Temp < 18 OR Temp > 25`. (Also correct: `NOT (Temp >= 18 AND Temp <= 25)`.)

```
DECLARE Temp : INTEGER
Temp ← 27
OUTPUT 29 DIV 4, " ", 29 MOD 4
OUTPUT Temp < 18 OR Temp > 25
```

</details>

Common mistakes

- `IF 13 <= Age <= 19`. Each comparison needs its own variable: `Age >= 13 AND Age <= 19`.
- `Age >= 13 AND <= 19`. The second half has no left side. Write `Age` again.
- `!=` or `==` for "not equal" and "equal". Those are Python and C. Pseudocode uses `<>` and `=`.

exercises.lol exercises.com.cn

4 Selection

4.1 ■ IF, THEN, ELSE, ENDIF

Selection means the program chooses what to do next, depending on a condition. A condition is an expression that is `TRUE` or `FALSE`.

4.1.1 IF without ELSE

```
IF <condition> THEN
   <statement(s)>
ENDIF
```

```
DECLARE Temperature : INTEGER
Temperature ← 31
IF Temperature > 30 THEN
   OUTPUT "Hot day"
   OUTPUT "Drink water"
ENDIF
OUTPUT "Have a good day"
```

If the condition is `TRUE`, the statements between `THEN` and `ENDIF` run. If it is `FALSE`, they are skipped. Either way, the program then carries on after `ENDIF`.

4.1.2 IF with ELSE

```
IF <condition> THEN
   <statement(s)>
ELSE
   <statement(s)>
ENDIF
```

```
DECLARE Mark : INTEGER
Mark ← 41
IF Mark >= 50 THEN
   OUTPUT "Pass"
ELSE
   OUTPUT "Fail"
ENDIF
```

Exactly **one** of the two branches runs: never both, never neither.

4.1.3 The 9618 layout

- `THEN` goes at the **end of the `IF` line**.
- `ELSE` and `ENDIF` line up with the `IF`.
- The statements in each branch are indented.

exercises.lol exercises.com.cn

Common mistakes

- Forgetting `ENDIF`. Every `IF` needs one.
- `IF Mark ← 50 THEN`. The arrow stores a value; a condition compares with `=`.
- Writing `THEN` on its own line below the `IF`. That is the **IGCSE 0478** layout. In 9618 it goes on the `IF` line.

4.2 ■ Nested IF

An `IF` inside another `IF` is called a **nested** `IF`. Use it when there are more than two outcomes, or when a second question only makes sense after the first.

```
DECLARE Mark : INTEGER
Mark ← 64
IF Mark >= 70 THEN
   OUTPUT "Distinction"
ELSE
   IF Mark >= 50 THEN
      OUTPUT "Merit"
   ELSE
      OUTPUT "Fail"
   ENDIF
ENDIF
```

- The inner `IF` sits inside the outer `ELSE`, and it has its **own** `ENDIF`.
- Count them: two `IF`s need two `ENDIF`s.
- **Test the highest boundary first.** The inner test `Mark >= 50` only runs when the mark is already below 70, so it does not need `AND Mark < 70`.

There is **no** `ELSEIF` or `ELIF` keyword in Cambridge pseudocode. Write `ELSE`, then a new `IF` inside it, as above.

4.2.1 A second question inside the first

Sometimes the inner `IF` sits in the `THEN` branch:

```
DECLARE ChallengerScore : INTEGER
DECLARE ChampionScore : INTEGER
DECLARE HighestScore : INTEGER
ChallengerScore ← 95
ChampionScore ← 80
HighestScore ← 90
IF ChallengerScore > ChampionScore THEN
   IF ChallengerScore > HighestScore THEN
      OUTPUT "New champion and new highest score"
   ELSE
      OUTPUT "New champion"
   ENDIF
ELSE
   OUTPUT "The champion stays"
ENDIF
```

Common mistakes

- Missing an inner `ENDIF`. Then every line after it seems to belong to the wrong block.
- Testing the smallest boundary first: with `IF Mark >= 50` first, a mark of 85 prints `Merit` and never reaches the `Distinction` test.
- Writing `ELSE IF` and closing it with only one `ENDIF`. Each `IF` closes separately.

4.3 ■ CASE OF

`CASE OF` chooses one branch by the **value of one variable**. It is clearer than many nested `IF`s when you test the same variable again and again.

```
CASE OF <identifier>
   <value 1> : <statement(s)>
   <value 2> : <statement(s)>
   OTHERWISE : <statement(s)>
ENDCASE
```


exercises.lol exercises.com.cn

```
DECLARE Move : CHAR
DECLARE Position : INTEGER
Position ← 50
Move ← 'A'
CASE OF Move
   'W' : Position ← Position - 10
   'S' : Position ← Position + 10
   'A' : Position ← Position - 1
   'D' : Position ← Position + 1
   OTHERWISE : OUTPUT "Unknown key"
ENDCASE
OUTPUT Position
```

- Each line is a **value**, a colon `:`, then what to do.
- The cases are tested **in order**. The first one that matches runs, and then the whole `CASE` is finished. The rest are not tested.
- `OTHERWISE` is optional. If it is there, it must be **last**, and it runs when no value matched. In 9618 it takes a colon: `OTHERWISE :`.

4.3.1 Ranges with TO

In 9618 a case can be a **range** range : `<value1> TO <value2>`. Both ends are included.

```
DECLARE Mark : INTEGER
Mark ← 64
CASE OF Mark
   70 TO 100 : OUTPUT "A"
   60 TO 69 : OUTPUT "B"
   50 TO 59 : OUTPUT "C"
   OTHERWISE : OUTPUT "U"
ENDCASE
```

4.3.2 Several statements in one case

A case may run more than one statement. Put them on the following lines, indented:

```
DECLARE Choice : INTEGER
DECLARE Balance : REAL
Balance ← 100.0
Choice ← 2
CASE OF Choice
   1 : OUTPUT "Balance: ", Balance
   2 :
      Balance ← Balance - 20.0
      OUTPUT "Took out 20. Balance: ", Balance
   OTHERWISE : OUTPUT "Not a menu choice"
ENDCASE
```

4.3.3 IF or CASE?

- One variable compared with **fixed values or ranges**: `CASE OF` is often clearer.
- A condition with **more than one variable**, or with `AND` / `OR`: you need `IF`.

4.3.4 Practice

1. `Day` holds a number from 1 to 7. Using `CASE OF`, output `"Weekday"` for 1 to 5, `"Weekend"` for 6 or 7, and `"Error"` for anything else.

2. Output `"Child"` for ages 0–12, `"Teen"` for 13–19 and `"Adult"` for 20 or more, using nested `IF`.

<details> <summary>Model answer</summary>

```
DECLARE Day : INTEGER
Day ← 6
CASE OF Day
   1 TO 5 : OUTPUT "Weekday"
   6 TO 7 : OUTPUT "Weekend"
   OTHERWISE : OUTPUT "Error"
ENDCASE
```


exercises.lol exercises.com.cn

```
DECLARE Age : INTEGER
Age ← 15
IF Age <= 12 THEN
   OUTPUT "Child"
ELSE
   IF Age <= 19 THEN
      OUTPUT "Teen"
   ELSE
      OUTPUT "Adult"
   ENDIF
ENDIF
```

</details>

Common mistakes

- Using `CASE OF` with a condition like `Mark >= 50 :`. A case is a value or a range, never a comparison.
- Putting `OTHERWISE` anywhere but last.
- Forgetting `ENDCASE`.
- `OTHERWISE` with no colon. That is the IGCSE form; 9618 writes `OTHERWISE :`.

5 Iteration

5.1 ■ FOR loops and STEP

Iteration means repeating a group of statements. A **loop** is the structure that does the repeating. Pseudocode has three loops, and each one fits a different job.

5.1.1 The count-controlled loop

Use `FOR` when you know **how many times** to repeat before the loop starts. It is called a **count-controlled loop**.

```
FOR <identifier> ← <value1> TO <value2>
   <statement(s)>
NEXT <identifier>
```

```
DECLARE Counter : INTEGER
FOR Counter ← 1 TO 5
   OUTPUT Counter, " x 3 = ", Counter * 3
NEXT Counter
```

- The loop counter (`Counter` here) must be an `INTEGER`.
- It takes **every** value from `value1` to `value2`, **including both ends**. So `1 TO 5` runs five times.
- If `value1 = value2`, the body runs once. If `value1 > value2`, it does not run at all.
- Writing the counter's name after `NEXT` is good practice, especially with one loop inside another.
- Do not change the counter yourself inside the loop. The `FOR` loop does that.

5.1.2 STEP

Add `STEP` to count in jumps other than 1. The step can be **negative** to count down.

```
FOR <identifier> ← <value1> TO <value2> STEP <increment>
```

```
DECLARE i : INTEGER
FOR i ← 0 TO 20 STEP 5
   OUTPUT i
NEXT i
FOR i ← 3 TO 1 STEP -1
   OUTPUT "Countdown ", i
NEXT i
```

The loop stops as soon as the next value would go **past** `value2`. So `FOR i ← 1 TO 10 STEP 4` gives 1, 5 and 9 only.

5.1.3 Loops inside loops

A loop can sit inside another loop. The inner loop runs **completely** for each single pass of the outer loop.

```
DECLARE Row : INTEGER
DECLARE Column : INTEGER
DECLARE Line : STRING
FOR Row ← 1 TO 3
   Line ← ""
   FOR Column ← 1 TO 4
      Line ← Line & NUM_TO_STR(Row * Column) & " "
   NEXT Column
   OUTPUT Line
NEXT Row
```

(`NUM_TO_STR` turns a number into a string so that `&` can join it; see topic 7.) The outer loop runs 3 times and the inner loop runs 4 times each time: 12 passes in total.

Common mistakes

- `FOR Counter = 1 TO 5`. The `FOR` line uses the arrow `←`.
- Counting down with no `STEP -1`. `FOR i ← 10 TO 1` does not run at all.
- Ending the inner and outer loops in the wrong order. The loop that opened **last** must close **first**.

5.2 ■ WHILE, the pre-condition loop

Use `WHILE` when you do **not** know how many times to repeat, and the loop may need to run **zero** times.

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

```
DECLARE Number : INTEGER
Number ← 40
WHILE Number > 9
   Number ← Number - 9
ENDWHILE
OUTPUT Number
```

- The condition is tested **before** each pass. That is why it is called a **pre-condition loop**.
- The loop repeats **while** the condition is `TRUE`, and stops when it becomes `FALSE`.
- If the condition is `FALSE` the first time, the body **never runs**.

exercises.lol exercises.com.cn

5.2.1 Something inside must change

The body must change something that the condition tests. If it does not, the condition stays `TRUE` for ever and you get an **infinite loop** .

```
DECLARE Balance : REAL
DECLARE Years : INTEGER
Balance ← 1000.0
Years ← 0
WHILE Balance < 2000.0
   Balance ← Balance * 1.1
   Years ← Years + 1
ENDWHILE
OUTPUT "Doubled after ", Years, " years"
```

Here `Balance` grows on every pass, so the loop must end.

Common mistakes

- Forgetting `ENDWHILE`.
- Not setting the variable before the loop. The condition is tested first, so the variable needs a value already.
- An infinite loop: nothing in the body changes the condition.

5.3 ■ REPEAT, the post-condition loop

Use `REPEAT` when the body must run **at least once**, for example to ask the user for input and then check it.

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

```
DECLARE Mark : INTEGER
REPEAT
   OUTPUT "Enter a mark from 0 to 100"
   INPUT Mark
UNTIL Mark >= 0 AND Mark <= 100
OUTPUT "Mark saved: ", Mark
```

- The condition is tested **after** each pass: a **post-condition loop** .
- The loop repeats **until** the condition is `TRUE`. When it becomes `TRUE`, the loop stops.
- There is no `ENDREPEAT`: the `UNTIL` line closes the loop.

exercises.lol exercises.com.cn

Notice that the two conditions are opposites: `WHILE` keeps going while its condition is `TRUE`; `REPEAT` stops when its condition is `TRUE`.

```
DECLARE Total : INTEGER
DECLARE Count : INTEGER
Total ← 0
Count ← 0
REPEAT
   Count ← Count + 1
   Total ← Total + Count
UNTIL Total > 20
OUTPUT "Added 1 to ", Count, " to pass 20: total ", Total
```

Common mistakes

- Writing the condition the wrong way round. “Keep asking while the age is wrong” becomes `UNTIL` the age is **right**.
- Using `REPEAT` when the body might need to run zero times. Then use `WHILE`.

5.4 ■ Choosing the right loop

Loop	Use it when	Tested	Runs at least
`FOR`	you know the number of passes before you start	(counts)	0 times
`WHILE`	you do not know, and it may need **zero** passes	**before** each pass	0 times
`REPEAT`	you do not know, and it needs **at least one** pass	**after** each pass	1 time

Typical exam situations:

- Go through every element of an array: `FOR`.
- Read a file until it ends: `WHILE NOT EOF(...)`, because the file may be empty.
- Ask for input until it is valid: `REPEAT ... UNTIL`.
- Search an array and stop early once you find the item: `WHILE` (or `REPEAT`) with a flag.

5.4.1 Practice

1. Output the even numbers from 2 to 20 with one `FOR` loop.

2. Starting at 1, keep doubling a number. Output how many doublings it takes to go above 1000. Use a `WHILE` loop.

3. Rewrite this `WHILE` loop as a `REPEAT` loop that gives the same output:

```
Count ← 1
WHILE Count <= 3
   OUTPUT Count
   Count ← Count + 1
ENDWHILE
```

<details> <summary>Model answer</summary>

```
DECLARE Number : INTEGER
FOR Number ← 2 TO 20 STEP 2
   OUTPUT Number
NEXT Number
```

```
DECLARE Value : INTEGER
DECLARE Doublings : INTEGER
Value ← 1
Doublings ← 0
WHILE Value <= 1000
   Value ← Value * 2
   Doublings ← Doublings + 1
ENDWHILE
OUTPUT Doublings
```

It prints `10`, because 2 to the power 10 is 1024.

```
DECLARE Count : INTEGER
Count ← 1
REPEAT
   OUTPUT Count
   Count ← Count + 1
UNTIL Count > 3
```

The `WHILE` condition `Count <= 3` turns into its opposite, `Count > 3`. The two loops only give the same output because the `WHILE` body runs at least once here.

</details>

exercises.lol exercises.com.cn

6 Arrays

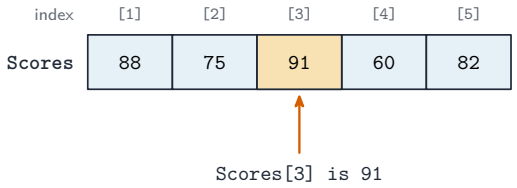
6.1 ■ One-dimensional arrays

An **array** stores many values of the **same data type** under one name. Each value is an **element**, and you reach it by its **index**: its position number. An array has a fixed size that does not change.

```
DECLARE <identifier> : ARRAY[<lower>:<upper>] OF <data type>
```

```
DECLARE Scores : ARRAY[1:5] OF INTEGER
Scores[1] ← 88
Scores[2] ← 75
Scores[3] ← 91
Scores[4] ← 60
Scores[5] ← 82
OUTPUT "The third score is ", Scores[3]
```

- `[1:5]` gives the **lower bound** lower bound 1 and the **upper bound** upper bound 5. So this array has 5 elements.
- Cambridge usually starts at **1**. Always write the lower bound, because some languages start at 0.
- The index goes in **square brackets**: `Scores[3]`.
- The index can be any expression that gives a whole number: `Scores[i]`, `Scores[n + 1]`.



Scores[1:5]: five boxes, one name, each reached by its index

6.1.1 Use a loop to visit every element

A `FOR` loop with the counter as the index visits each element in turn. This is the most common pattern in Paper 2.

```
DECLARE Names : ARRAY[1:4] OF STRING
DECLARE i : INTEGER
Names[1] ← "Ali"
Names[2] ← "Bo"
Names[3] ← "Chen"
Names[4] ← "Dara"
FOR i ← 1 TO 4
   OUTPUT i, ": ", Names[i]
NEXT i
```

You **cannot** fill a group of elements in one statement. `Names[1 TO 4] ← ""` is not allowed. Use a loop:

```
DECLARE Names : ARRAY[1:30] OF STRING
DECLARE Index : INTEGER
FOR Index ← 1 TO 30
   Names[Index] ← ""
NEXT Index
OUTPUT "All 30 names cleared"
```

You **can** copy a whole array to another array of the same size and type in one line: `SavedGame ← Board`.

Common mistakes

- An index outside the bounds, such as `Scores[6]` or `Scores[0]` for `ARRAY[1:5]`. This is a run-time error.
- Using round brackets: `Scores(3)`. Arrays use square brackets.
- Forgetting the data type: every array needs `OF <data type>`.

6.2 ■ Two-dimensional arrays

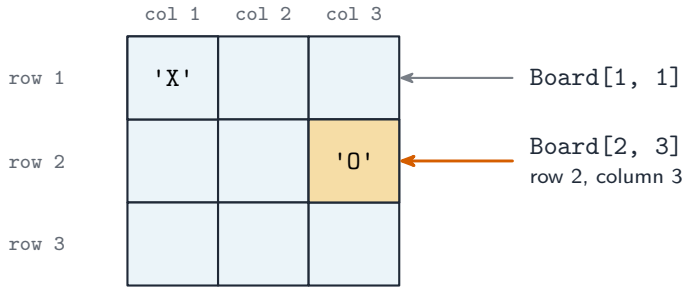
A **two-dimensional** two-dimensional array is like a table, with **rows** and **columns**. You need two indices to reach one element.

```
DECLARE <identifier> : ARRAY[<lower1>:<upper1>, <lower2>:<upper2>] OF
↪  <data type>
```

```
DECLARE Board : ARRAY[1:3, 1:3] OF CHAR
Board[1, 1] ← 'X'
Board[2, 3] ← 'O'
OUTPUT "Row 2, column 3 holds ", Board[2, 3]
```


exercises.lol exercises.com.cn

- By convention the first index is the **row** and the second is the **column**.
- Both indices go inside **one** pair of square brackets, separated by a comma: `Board[2, 3]`, not `Board[2][3]`.



Board[1:3, 1:3]: the first index picks the row, the second the column

6.2.1 Nested loops for a 2D array

To visit every element, use one loop for the rows and another loop inside it for the columns.

```
DECLARE Amount : ARRAY[1:2, 1:3] OF INTEGER
DECLARE Row : INTEGER
DECLARE Column : INTEGER
DECLARE RowTotal : INTEGER
Amount[1, 1] ← 4
Amount[1, 2] ← 7
Amount[1, 3] ← 1
Amount[2, 1] ← 9
Amount[2, 2] ← 2
Amount[2, 3] ← 5
FOR Row ← 1 TO 2
   RowTotal ← 0
   FOR Column ← 1 TO 3
      RowTotal ← RowTotal + Amount[Row, Column]
   NEXT Column
   OUTPUT "Total for row ", Row, " is ", RowTotal
NEXT Row
```

`RowTotal ← 0` is **inside** the outer loop, so each row starts again from zero.

Common mistakes

- Swapping the indices: `Board[Column, Row]` when the array was set up as rows first.

- Setting `RowTotal ← 0` **before** the outer loop. Then the second row's total also contains the first row.

6.3 ■ Totals, counts, largest and smallest

Most array questions ask for one of a few patterns. Learn them as shapes.

6.3.1 Total and average

```
DECLARE Marks : ARRAY[1:5] OF INTEGER
DECLARE Total : INTEGER
DECLARE i : INTEGER
Marks[1] ← 56
Marks[2] ← 71
Marks[3] ← 48
Marks[4] ← 90
Marks[5] ← 65
Total ← 0
FOR i ← 1 TO 5
   Total ← Total + Marks[i]
NEXT i
OUTPUT "Total: ", Total
OUTPUT "Average: ", Total / 5
```

Set the **total** to 0 **before** the loop, add inside it, and divide **after** it.

6.3.2 Count the elements that pass a test

```
DECLARE Marks : ARRAY[1:5] OF INTEGER
DECLARE PassCount : INTEGER
DECLARE i : INTEGER
Marks[1] ← 56
Marks[2] ← 71
Marks[3] ← 48
Marks[4] ← 90
Marks[5] ← 65
PassCount ← 0
FOR i ← 1 TO 5
   IF Marks[i] >= 60 THEN
      PassCount ← PassCount + 1
   ENDIF
NEXT i
OUTPUT PassCount, " marks are 60 or more"
```


exercises.lol exercises.com.cn

6.3.3 Largest and smallest

Start with the **first element**, not with 0. Then compare every other element with it.

```
DECLARE Marks : ARRAY[1:5] OF INTEGER
DECLARE Highest : INTEGER
DECLARE Lowest : INTEGER
DECLARE i : INTEGER
Marks[1] ← 56
Marks[2] ← 71
Marks[3] ← 48
Marks[4] ← 90
Marks[5] ← 65
Highest ← Marks[1]
Lowest ← Marks[1]
FOR i ← 2 TO 5
   IF Marks[i] > Highest THEN
      Highest ← Marks[i]
   ENDIF
   IF Marks[i] < Lowest THEN
      Lowest ← Marks[i]
   ENDIF
NEXT i
OUTPUT "Highest ", Highest, ", lowest ", Lowest
```

Why not start `Lowest` at 0? Because no mark is below 0, so `Lowest` would stay 0 and be wrong. Starting with `Marks[1]` always works, even with negative numbers.

6.3.4 Practice

1. An array `Temps[1:7]` holds a week of temperatures. Output the number of the day with the highest temperature (not the temperature itself).

<details> <summary>Model answer</summary>

```
DECLARE Temps : ARRAY[1:7] OF INTEGER
DECLARE HottestDay : INTEGER
DECLARE Day : INTEGER
Temps[1] ← 21
Temps[2] ← 25
Temps[3] ← 19
Temps[4] ← 28
Temps[5] ← 24
Temps[6] ← 28
Temps[7] ← 22
HottestDay ← 1
FOR Day ← 2 TO 7
   IF Temps[Day] > Temps[HottestDay] THEN
      HottestDay ← Day
   ENDIF
NEXT Day
OUTPUT "Hottest day: ", HottestDay
```

It prints 4. Store the **index** of the best element so far, not its value. Day 6 is also 28, but `>` does not replace an equal value, so the first hottest day is kept.

</details>

Common mistakes

- Setting `Total ← 0` inside the loop. Then it restarts on every pass.
- Dividing by the wrong number for the average. Divide by how many values were added.
- Using `>=` when the question wants the **first** best value. On a tie `>=` replaces the stored value again, so you end up with the **last** best one.

exercises.lol exercises.com.cn

7 Strings and built-in functions

7.1 ■ Joining strings with &

To **concatenate** means to join strings end to end. In 9618 the operator is `&`.

```
DECLARE FirstName : STRING
DECLARE LastName : STRING
DECLARE FullName : STRING
FirstName ← "Summer"
LastName ← "Pudding"
FullName ← FirstName & " " & LastName
OUTPUT FullName
OUTPUT LENGTH(FullName)
```

- `&` makes **one new string**. You can store it, pass it to a function, or output it.
- Nothing is added between the parts. To get a space, join `" "` yourself.
- `&` joins strings and characters. To join a number, first turn it into a string with `NUM_TO_STR` (subtopic 7.4).

7.1.1 Building a string in a loop

Start with the empty string and add to it on each pass:

```
DECLARE Stars : STRING
DECLARE i : INTEGER
Stars ← ""
FOR i ← 1 TO 5
   Stars ← Stars & "*"
NEXT i
OUTPUT Stars
```

7.1.2 & or a comma?

- In `OUTPUT`, a comma prints values one after another: `OUTPUT "Age: ", Age`. The values can be any type.
- `&` makes a new **string value**, for example to store in a variable or write to a file.

Common mistakes

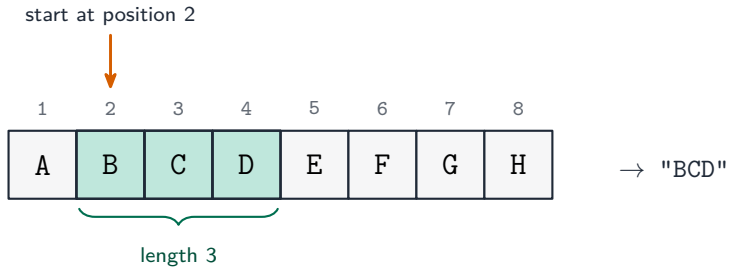
- `"Total: " & 25`. 25 is a number; write `"Total: " & NUM_TO_STR(25)`.
- `+` to join strings, as in Python. Pseudocode uses `&`.

7.2 ■ LEFT, RIGHT, MID and LENGTH

In the exam, the string functions you may use are **always printed in an insert** with the paper. You still need to know how they behave, because you must use them quickly and correctly. Positions in a string start at **1**.

Function	Returns	Example	Result
`LENGTH(S)`	the number of characters in S	`LENGTH("Happy Days")`	`10`
`LEFT(S, x)`	the first x characters	`LEFT("ABCDEFGH", 3)`	`"ABC"`
`RIGHT(S, x)`	the last x characters	`RIGHT("ABCDEFGH", 3)`	`"FGH"`
`MID(S, x, y)`	y characters, starting at position x	`MID("ABCDEFGH", 2, 3)`	`"BCD"`

- `LENGTH` counts spaces and punctuation too.
- In `MID`, the third value is a **length**, not an end position. `MID("ABCDEFGH", 2, 3)` starts at B and takes 3 characters.
- A **substring** is part of a string. `LEFT`, `RIGHT` and `MID` all return one.



MID("ABCDEFGH", 2, 3): start at position 2 and take 3 characters

exercises.lol exercises.com.cn

7.2.1 Visit each character

`MID(S, i, 1)` is the character at position `i`. A loop from 1 to `LENGTH(S)` visits them all. This pattern appears in almost every string question.

```
DECLARE Word : STRING
DECLARE i : INTEGER
DECLARE Vowels : INTEGER
DECLARE Letter : CHAR
Word ← "EDUCATION"
Vowels ← 0
FOR i ← 1 TO LENGTH(Word)
   Letter ← MID(Word, i, 1)
   IF Letter = 'A' OR Letter = 'E' OR Letter = 'I' OR Letter = 'O' OR
   ↪  Letter = 'U' THEN
      Vowels ← Vowels + 1
   ENDIF
NEXT i
OUTPUT Word, " has ", Vowels, " vowels"
```

7.2.2 Reverse a string

```
DECLARE Word : STRING
DECLARE Reversed : STRING
DECLARE i : INTEGER
Word ← "STRESSED"
Reversed ← ""
FOR i ← LENGTH(Word) TO 1 STEP -1
   Reversed ← Reversed & MID(Word, i, 1)
NEXT i
OUTPUT Reversed
```

Common mistakes

- Starting at position 0. The first character is at position **1**.
- `MID(S, 2, 5)` meaning "from 2 to 5". It means "5 characters from position 2".
- `SUBSTRING(S, 2, 3)`. That is the IGCSE name. In 9618 the function is `MID`.
- Asking for more characters than there are: `LEFT("Hi", 5)` is an error.

7.3 ■ Case and character codes

7.3.1 Upper and lower case

Function	Returns	Example	Result
`TO_UPPER(x)`	x with every letter in upper case; x is a `STRING` or a `CHAR`	`TO_UPPER("Hello")`	`"HELLO"`
`TO_LOWER(x)`	x with every letter in lower case	`TO_LOWER("Hello")`	`"hello"`
`UCASE(c)`	one `CHAR` in upper case	`UCASE('h')`	`'H'`
`LCASE(c)`	one `CHAR` in lower case	`LCASE('W')`	`'w'`

Characters that are not letters, such as digits and spaces, do not change. `TO_UPPER` and `TO_LOWER` are the ones printed in the Paper 2 insert, and they work on whole strings.

A common use: compare two strings **without caring about case**.

```
DECLARE Answer : STRING
Answer ← "yEs"
IF TO_UPPER(Answer) = "YES" THEN
   OUTPUT "You said yes"
ENDIF
```

7.3.2 Character codes: ASC and CHR

Each character is stored as a number, its **character code** . In ASCII, `'A'` is 65, `'B'` is 66, … `'Z'` is 90, and `'a'` is 97.

Function	Returns	Example	Result
`ASC(c)`	the code of the character c	`ASC('A')`	`65`
`CHR(x)`	the character with code x	`CHR(66)`	`'B'`

```
DECLARE Letter : CHAR
DECLARE Code : INTEGER
Letter ← 'C'
Code ← ASC(Letter)
OUTPUT Letter, " has code ", Code
OUTPUT "The next letter is ", CHR(Code + 1)
```

Two things the codes make easy:

- **Is it a capital letter?** `Letter >= 'A' AND Letter <= 'Z'` works because the codes are in order.
- **Shift a letter** (as in a Caesar cipher): `CHR(ASC(Letter) + 3)`.

Common mistakes

exercises.lol exercises.com.cn

- `UCASE("hello")` on a whole string in a 9618 answer. `UCASE` takes one character; use `TO_UPPER` for a string.
- Forgetting that `'a'` and `'A'` have different codes, so `'a' = 'A'` is `FALSE`.

7.4 ■ Numbers and strings

7.4.1 Converting between them

Function	Returns	Example	Result
`NUM_TO_STR(x)`	the number x as a string	`NUM_TO_STR(87.5)`	`"87.5"`
`STR_TO_NUM(x)`	the string x as a number	`STR_TO_NUM("23")`	`23`
`IS_NUM(S)`	`TRUE` if S is a valid number	`IS_NUM("12a")`	`FALSE`

Data read from a file or typed by a user often arrives as a string. Check it with `IS_NUM` before you convert it, or the program can fail.

```
DECLARE Entry : STRING
DECLARE Value : INTEGER
Entry ← "42"
IF IS_NUM(Entry) THEN
   Value ← STR_TO_NUM(Entry)
   OUTPUT "Double: ", Value * 2
ELSE
   OUTPUT "Not a number"
ENDIF
OUTPUT "Code " & NUM_TO_STR(7) & "B"
```

7.4.2 INT and RAND

Function	Returns	Example	Result
`INT(x)`	the whole-number part of x; it **cuts off** the decimals	`INT(27.9)`	`27`
`RAND(x)`	a random `REAL` from 0 up to x, **not including** x	`RAND(87)`	e.g. `35.43`

`INT` does **not** round: `INT(27.9)` is 27, not 28. To round to the nearest whole number, add 0.5 first: `INT(X + 0.5)`.

To get a random whole number from 1 to 6, like a dice:

```
DECLARE Dice : INTEGER
Dice ← INT(RAND(6)) + 1
OUTPUT "You rolled ", Dice
```

`RAND(6)` gives a value from 0 up to (not including) 6. `INT` makes that 0 to 5, and `+ 1`

gives 1 to 6.

7.4.3 Dates

`DATE` values have their own functions in the insert, such as `DAY(ThisDate)`, `MONTH(ThisDate)`, `YEAR(ThisDate)` and `SETDATE(Day, Month, Year)`. Read the insert in your paper for the exact list: it describes each one.

7.4.4 Practice

1. A product code such as `"AB1234"` has two letters and then four digits. Write pseudocode that outputs `"Valid"` if the last four characters are a number, and `"Invalid"` otherwise.

2. Output the initials of `"Grace Brewster Hopper"`: the first letter of each word.

<details> <summary>Model answer</summary>

```
DECLARE Code : STRING
Code ← "AB1234"
IF LENGTH(Code) = 6 AND IS_NUM(RIGHT(Code, 4)) THEN
   OUTPUT "Valid"
ELSE
   OUTPUT "Invalid"
ENDIF
```

```
DECLARE FullName : STRING
DECLARE Initials : STRING
DECLARE i : INTEGER
FullName ← "Grace Brewster Hopper"
Initials ← LEFT(FullName, 1)
FOR i ← 2 TO LENGTH(FullName)
   IF MID(FullName, i - 1, 1) = " " THEN
      Initials ← Initials & MID(FullName, i, 1)
   ENDIF
NEXT i
OUTPUT Initials
```

It prints `GBH`. A letter starts a new word when the character **before** it is a space.

</details>

Common mistakes

- Doing arithmetic on a string: `"5" + 1`. Convert first with `STR_TO_NUM`.
- Thinking `INT` rounds. It always cuts off the fractional part.
- `RAND(6)` used as a dice. It gives decimals like 3.81 and can give 0; use `INT(RAND(6)) + 1`.

exercises.lol exercises.com.cn

8 Procedures and functions

8.1 ■ PROCEDURE and CALL

A **subroutine** is a named block of code that does one job. You write it once and use it as often as you need. Pseudocode has two kinds: procedures and functions.

A **procedure** carries out a task. It does not give a value back.

```
PROCEDURE <identifier>()
   <statement(s)>
ENDPROCEDURE

PROCEDURE <identifier>(<param1> : <data type>, <param2> : <data type>
↪  ...)
   <statement(s)>
ENDPROCEDURE
```

You run a procedure with `CALL`:

```
CALL <identifier>()
CALL <identifier>(<value1>, <value2> ...)
```

```
PROCEDURE DrawLine(LineLength : INTEGER, Symbol : CHAR)
   DECLARE Line : STRING
   DECLARE i : INTEGER
   Line ← ""
   FOR i ← 1 TO LineLength
      Line ← Line & Symbol
   NEXT i
   OUTPUT Line
ENDPROCEDURE

PROCEDURE ShowTitle()
   CALL DrawLine(12, '=')
   OUTPUT "  MAIN MENU"
   CALL DrawLine(12, '=')
ENDPROCEDURE

CALL ShowTitle()
CALL DrawLine(5, '*')
```

- A **parameter** is a variable in the header that receives a value. `LineLength` and `Symbol` are the parameters of `DrawLine`.
- The value you send in the `CALL` is called an **argument**. The arguments must be in the **same order** and of the **right types** as the parameters.
- A procedure with no parameters still has empty brackets: `ShowTitle()`.

- A `CALL` is a complete statement on its own line.

8.1.1 Why use subroutines?

- The same code is not written again and again.
- Each subroutine can be written and tested on its own.
- The main program becomes short and easy to read.

Common mistakes

- Calling a procedure without `CALL`: write `CALL DrawLine(5, '*')`, not just `DrawLine(5, '*')`.
- Leaving out the data type of a parameter in the header.
- Forgetting `ENDPROCEDURE`.

8.2 ■ FUNCTION and RETURN

A **function** works like a procedure, but it **returns a single value** to the place where it was used. The header says the data type of that value.

```
FUNCTION <identifier>(<param1> : <data type> ...) RETURNS <data type>
   <statement(s)>
ENDFUNCTION
```

```
FUNCTION Max(Number1 : INTEGER, Number2 : INTEGER) RETURNS INTEGER
   IF Number1 > Number2 THEN
      RETURN Number1
   ELSE
      RETURN Number2
   ENDIF
ENDFUNCTION

DECLARE Distance : INTEGER
Distance ← 8
OUTPUT "Penalty fine = ", Max(10, Distance * 2)
```

- `RETURN` gives the value back. When `RETURN` runs, the function **stops at once**; any lines after it are skipped.
- A function call is **not** a statement on its own. Use it **inside an expression**: in an assignment, an `OUTPUT`, a condition, or as an argument.
- **Never** use `CALL` with a function.

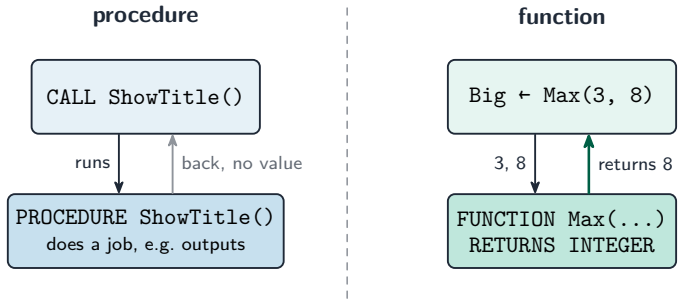
exercises.lol exercises.com.cn

```
FUNCTION IsEven(Number : INTEGER) RETURNS BOOLEAN
   RETURN Number MOD 2 = 0
ENDFUNCTION

DECLARE Value : INTEGER
FOR Value ← 1 TO 4
   IF IsEven(Value) THEN
      OUTPUT Value, " is even"
   ENDIF
NEXT Value
```

8.2.1 Procedure or function?

- The subroutine works out **one value** that the caller needs: a **function**.
- It **does** something (outputs, changes an array, writes a file) and has no single answer: a **procedure**.



A procedure is called to do a job; a function returns a value to use

Common mistakes

- `CALL Max(3, 4)`. A function is used in an expression: `Biggest ← Max(3, 4)`.
- Forgetting `RETURNS <data type>` in the header, or forgetting the `RETURN` statement in the body.
- Outputting the answer inside the function instead of returning it, when the question asks for a function that **returns** a value.

8.3 ■ BYVAL and BYREF

There are two ways to pass a value to a parameter.

- **By value** (`BYVAL`): the subroutine gets a **copy** of the value. If it changes the parameter, the caller's variable does **not** change.
- **By reference** (`BYREF`): the subroutine gets the caller's **own variable**. A change inside the subroutine **is** seen by the caller after the call.

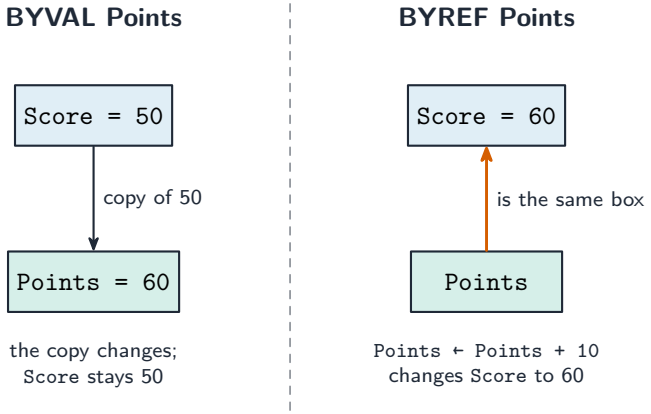
Write the keyword before the parameter in the header. If you write neither, the parameter is passed **by value**.

```
PROCEDURE AddBonus(BYVAL Points : INTEGER)
   Points ← Points + 10
ENDPROCEDURE

PROCEDURE AddBonusRef(BYREF Points : INTEGER)
   Points ← Points + 10
ENDPROCEDURE

DECLARE Score : INTEGER
Score ← 50
CALL AddBonus(Score)
OUTPUT "After BYVAL: ", Score
CALL AddBonusRef(Score)
OUTPUT "After BYREF: ", Score
```

The first `OUTPUT` shows 50 (only the copy changed). The second shows 60.



BYVAL gives the procedure a copy; BYREF gives it the caller's own variable

8.3.1 One keyword for several parameters

If several parameters in a row use the same method, write the keyword once. Here **both** `X` and `Y` are `BYREF`:

```
PROCEDURE Swap(BYREF X : INTEGER, Y : INTEGER)
   DECLARE Temp : INTEGER
   Temp ← X
   X ← Y
   Y ← Temp
ENDPROCEDURE

DECLARE A : INTEGER
DECLARE B : INTEGER
A ← 3
B ← 8
CALL Swap(A, B)
OUTPUT "A = ", A, ", B = ", B
```

8.3.2 When to use BYREF

- A procedure must give back **more than one** result.
- A procedure must change the caller's variable, as `Swap` does.

Do not pass parameters by reference to a **function**. A function gives back its one value with `RETURN`.

Common mistakes

- Expecting a change to a plain (`BYVAL`) parameter to be seen outside. It is a copy.
- Using `BYREF` in a `FUNCTION` header.
- Writing `BYREF` in an IGCSE 0478 answer. That syllabus does not have it.

8.4 ■ Local and global variables

The **scope** of a variable is the part of the program where it can be used.

- A **local variable** is declared **inside** a subroutine. It exists only while that subroutine runs, and only that subroutine can use it. Parameters are local too.
- A **global variable** is declared in the **main program**, outside every subroutine. Any part of the program can use it.

exercises.lol exercises.com.cn

```
DECLARE Total : INTEGER

PROCEDURE AddToTotal(Amount : INTEGER)
   DECLARE Doubled : INTEGER
   Doubled ← Amount * 2
   Total ← Total + Doubled
ENDPROCEDURE

Total ← 0
CALL AddToTotal(5)
CALL AddToTotal(3)
OUTPUT "Total: ", Total
```

- `Total` is **global**: the procedure can change it, and the main program sees the result, 16.
- `Doubled` and `Amount` are **local** to `AddToTotal`. The main program cannot use them.

8.4.1 Prefer local variables

- A subroutine that uses only its parameters and local variables can be tested and reused on its own.
- Two subroutines can both use a local variable called `i` without any clash.
- A global that many subroutines change makes errors hard to find.

So pass data in through **parameters**, give results back with `RETURN` (or `BYREF`), and keep globals for the few values that really belong to the whole program.

8.4.2 Practice

1. Write a function `Area(Width : REAL, Height : REAL) RETURNS REAL`, then use it to output the area of a 3.5 by 2.0 rectangle.

2. Write a procedure `MinMax` that takes three integers and gives back the smallest and the largest through two `BYREF` parameters.

<details> <summary>Model answer</summary>

```
FUNCTION Area(Width : REAL, Height : REAL) RETURNS REAL
   RETURN Width * Height
ENDFUNCTION

OUTPUT "Area: ", Area(3.5, 2.0)
```


exercises.lol exercises.com.cn

```
PROCEDURE MinMax(A : INTEGER, B : INTEGER, C : INTEGER, BYREF Smallest :
↪  INTEGER, Largest : INTEGER)
   Smallest ← A
   Largest ← A
   IF B < Smallest THEN
      Smallest ← B
   ENDIF
   IF C < Smallest THEN
      Smallest ← C
   ENDIF
   IF B > Largest THEN
      Largest ← B
   ENDIF
   IF C > Largest THEN
      Largest ← C
   ENDIF
ENDPROCEDURE

DECLARE Low : INTEGER
DECLARE High : INTEGER
CALL MinMax(7, 2, 9, Low, High)
OUTPUT "Smallest ", Low, ", largest ", High
```

`A`, `B` and `C` are passed by value (the default). `BYREF` before `Smallest` also covers `Largest`, because the keyword is not repeated.

</details>

9 Records and user-defined types

9.1 ■ Defining a record type

An array holds many values of **one** type. Often you need to keep **different** types together: a student's name (`STRING`), mark (`INTEGER`) and form group (`CHAR`). A **record** does this. Each item inside a record is a **field**.

A record is a **user-defined data type**: you design it first, then declare variables of that type, just as you would with `INTEGER`.

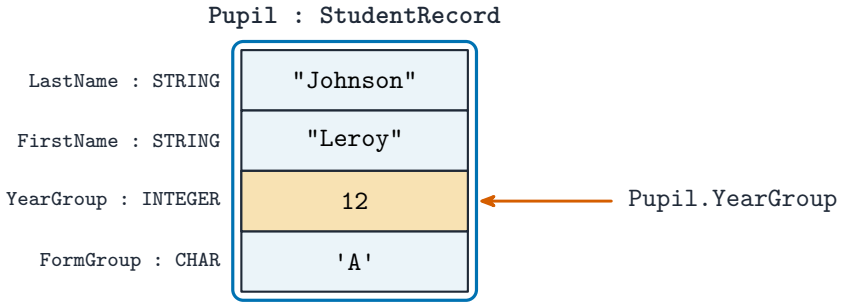
```
TYPE <identifier>
   DECLARE <field1> : <data type>
   DECLARE <field2> : <data type>
   ...
ENDTYPE
```

```
TYPE StudentRecord
   DECLARE LastName : STRING
   DECLARE FirstName : STRING
   DECLARE YearGroup : INTEGER
   DECLARE FormGroup : CHAR
ENDTYPE

DECLARE Pupil : StudentRecord
Pupil.LastName ← "Johnson"
Pupil.FirstName ← "Leroy"
Pupil.YearGroup ← 12
Pupil.FormGroup ← 'A'
OUTPUT Pupil.FirstName, " ", Pupil.LastName, " is in ", Pupil.YearGroup,
↪  Pupil.FormGroup
```

- `TYPE … ENDTYPE` only **describes** the record. It does not make a variable.
- `DECLARE Pupil : StudentRecord` makes a variable of the new type.
- Reach one field with **dot notation**: `<variable>.<field>`, as in `Pupil.YearGroup`.
- A field can be used anywhere a normal variable can: in `INPUT`, `OUTPUT`, conditions and assignments.

exercises.lol exercises.com.cn



A record groups fields of different types; dot notation reaches each one

9.1.1 Copying a whole record

Two variables of the same record type can be assigned in one line. Every field is copied.

```
TYPE Point
   DECLARE X : INTEGER
   DECLARE Y : INTEGER
ENDTYPE

DECLARE Start : Point
DECLARE Copy : Point
Start.X ← 3
Start.Y ← 7
Copy ← Start
Copy.X ← 10
OUTPUT "Start is (", Start.X, ", ", Start.Y, ")"
OUTPUT "Copy is (", Copy.X, ", ", Copy.Y, ")"
```

Changing `Copy.X` afterwards does not change `Start`: they are two separate records.

Common mistakes

- Using the **type** name instead of the variable: `StudentRecord.LastName ← "Lee"`. Write `Pupil.LastName`.
- `OUTPUT Pupil` to show a whole record. Output each field you need.
- Forgetting `DECLARE` in front of each field inside `TYPE … ENDTYPE`.

9.2 ■ Arrays of records

An array whose elements are records is a very common data structure: it is like a table where each row is one record and each column is one field.

```
TYPE Book
   DECLARE Title : STRING
   DECLARE Pages : INTEGER
   DECLARE OnLoan : BOOLEAN
ENDTYPE

DECLARE Library : ARRAY[1:3] OF Book
DECLARE i : INTEGER
Library[1].Title ← "Dune"
Library[1].Pages ← 412
Library[1].OnLoan ← FALSE
Library[2].Title ← "Emma"
Library[2].Pages ← 474
Library[2].OnLoan ← TRUE
Library[3].Title ← "Holes"
Library[3].Pages ← 233
Library[3].OnLoan ← FALSE

FOR i ← 1 TO 3
   IF Library[i].OnLoan = FALSE THEN
      OUTPUT Library[i].Title, " (", Library[i].Pages, " pages) is
      ↪ available"
   ENDIF
NEXT i
```

- The index comes **first**, then the field: `Library[i].Title`.
- Read it from left to right: "in `Library`, element `i`, field `Title`".

9.2.1 Records instead of parallel arrays

Without records, you would need three separate arrays (`Titles`, `Pages`, `OnLoan`) that share one index. These are called **parallel arrays** . They work, but it is easy to sort one array and forget the others. An array of records keeps each book's data together.

Common mistakes

- `Library.Title[i]`. The index belongs to the array: `Library[i].Title`.
- Declaring the array before the `TYPE` that it uses. Define the type first.

exercises.lol exercises.com.cn

9.3 ■ Other user-defined types

The 9618 guide describes more user-defined types. They are **A-Level (Paper 3)** content, but you will see them in the same guide, so it helps to recognise them. They do not run on this page.

9.3.1 Enumerated types

An **enumerated type** lists every value a variable of that type may have.

```
TYPE Season = (Spring, Summer, Autumn, Winter)
DECLARE ThisSeason : Season
ThisSeason ← Spring
```

The values are names, **not** strings: there are no quotes around `Spring`.

9.3.2 Pointers and sets

- A **pointer** type holds a memory address. `TYPE TIntPointer = ^INTEGER` declares it; `^` before a variable gives its address, and `^` after a pointer gives the value it points to.
- A **set** type holds a group of values with no order: `TYPE LetterSet = SET OF CHAR`.

For AS, focus on **records** and **arrays of records**: they appear in Paper 2 questions again and again.

9.3.3 Practice

1. Define a record type `Car` with the fields `Registration` (string), `Year` (integer) and `Price` (real). Declare an array of 3 cars, fill it, and output the registration of every car that costs less than 5000.

<details> <summary>Model answer</summary>

```
TYPE Car
   DECLARE Registration : STRING
   DECLARE Year : INTEGER
   DECLARE Price : REAL
ENDTYPE

DECLARE Cars : ARRAY[1:3] OF Car
DECLARE i : INTEGER
Cars[1].Registration ← "AB12 CDE"
Cars[1].Year ← 2015
Cars[1].Price ← 4500.0
Cars[2].Registration ← "FG34 HIJ"
Cars[2].Year ← 2019
Cars[2].Price ← 9800.0
Cars[3].Registration ← "KL56 MNO"
Cars[3].Year ← 2012
Cars[3].Price ← 2950.0
FOR i ← 1 TO 3
   IF Cars[i].Price < 5000.0 THEN
      OUTPUT Cars[i].Registration
   ENDIF
NEXT i
```

It prints `AB12 CDE` and `KL56 MNO`.

</details>

exercises.lol exercises.com.cn

10 Text files

10.1 ■ Opening and closing a file

Variables lose their values when a program ends. A **text file** keeps data on storage, so the next run can read it again. A text file is a list of **lines**, and each line is a string. Lines are read or written **in order**, one after another.

10.1.1 OPENFILE

Before you use a file, you must open it in one **file mode** :

```
OPENFILE <file identifier> FOR <file mode>
```

Mode	What it does
`READ`	read lines from the file, starting with the first line
`WRITE`	create a **new**, empty file and write to it; any old data in the file is **lost**
`APPEND`	add lines to the **end** of the file; the old data is kept

The file identifier is the file's name, as a string such as `"Scores.txt"`, or a `STRING` variable that holds the name.

10.1.2 CLOSEFILE

```
CLOSEFILE <file identifier>
```

Close every file when you have finished with it. A file is open in **one mode at a time**: to read a file you have just written, close it and open it again `FOR READ`.

```
DECLARE FileName : STRING
FileName ← "Log.txt"
OPENFILE FileName FOR WRITE
WRITEFILE FileName, "Program started"
CLOSEFILE FileName
OUTPUT "Written to ", FileName
```

Common mistakes

- Using a file without opening it first.
- Forgetting `CLOSEFILE`, often worth a mark in Paper 2.
- Opening with `WRITE` when you wanted to keep the old lines. `WRITE` empties the file; use `APPEND`.

10.2 ■ Writing and appending

`WRITEFILE` writes **one line** to a file that is open for `WRITE` or `APPEND`.

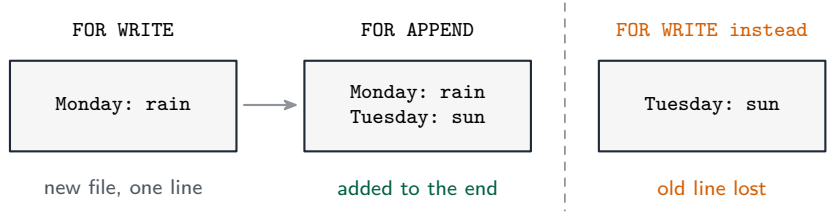
```
WRITEFILE <file identifier>, <data>
```

```
DECLARE Line : STRING
OPENFILE "Diary.txt" FOR WRITE
WRITEFILE "Diary.txt", "Monday: rain"
CLOSEFILE "Diary.txt"

OPENFILE "Diary.txt" FOR APPEND
WRITEFILE "Diary.txt", "Tuesday: sun"
CLOSEFILE "Diary.txt"

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

Both lines are printed, because `APPEND` kept the first one. Change `APPEND` to `WRITE` and run it again: only Tuesday is left.



WRITE starts an empty file; APPEND adds to the end

10.2.1 Several values on one line

A line is one string. To store several values, join them into one string with a separator separator , such as a comma:

```
DECLARE Name : STRING
DECLARE Score : INTEGER
Name ← "Ravi"
Score ← 87
OPENFILE "Results.txt" FOR WRITE
WRITEFILE "Results.txt", Name & "," & NUM_TO_STR(Score)
CLOSEFILE "Results.txt"
OUTPUT "Saved: ", Name & "," & NUM_TO_STR(Score)
```

10.2.2 Saving a whole array

Write one element per line inside a loop:

```
DECLARE Colours : ARRAY[1:3] OF STRING
DECLARE i : INTEGER
Colours[1] ← "red"
Colours[2] ← "green"
Colours[3] ← "blue"
OPENFILE "Colours.txt" FOR WRITE
FOR i ← 1 TO 3
   WRITEFILE "Colours.txt", Colours[i]
NEXT i
CLOSEFILE "Colours.txt"
OUTPUT "3 lines written"
```

Keep `OPENFILE` and `CLOSEFILE` **outside** the loop. Opening a file `FOR WRITE` inside the loop would empty it on every pass.

10.3 ■ Reading to the end of a file

`READFILE` reads the **next line** of a file that is open `FOR READ`, and stores it in a variable. The variable should be a `STRING`.

```
READFILE <file identifier>, <variable>
```

The function `EOF` tells you when there is nothing left to read. `EOF` stands for **end of file** :

```
EOF(<file identifier>)
```

`EOF` returns `TRUE` when there are no more lines (also when the file is empty), and `FALSE` otherwise.

exercises.lol exercises.com.cn

10.3.1 The standard reading loop

You usually do not know how many lines a file has, and it may be empty. So use a `WHILE` loop that tests `EOF` **before** each read:

```
DECLARE Line : STRING
DECLARE Count : INTEGER
OPENFILE "Names.txt" FOR WRITE
WRITEFILE "Names.txt", "Amir"
WRITEFILE "Names.txt", ""
WRITEFILE "Names.txt", "Bea"
CLOSEFILE "Names.txt"

Count ← 0
OPENFILE "Names.txt" FOR READ
WHILE NOT EOF("Names.txt")
   READFILE "Names.txt", Line
   IF Line <> "" THEN
      Count ← Count + 1
      OUTPUT Line
   ENDIF
ENDWHILE
CLOSEFILE "Names.txt"
OUTPUT Count, " names read"
```

The first part only creates the file, so that the example can run. The second part is the pattern to learn: **open, loop while not EOF, read, process, close**.

10.3.2 Copying one file to another

This is the guide's own example: copy every line, but replace blank lines with dashes.

```
DECLARE LineOfText : STRING
OPENFILE "FileA.txt" FOR READ
OPENFILE "FileB.txt" FOR WRITE
WHILE NOT EOF("FileA.txt")
   READFILE "FileA.txt", LineOfText
   IF LineOfText = "" THEN
      WRITEFILE "FileB.txt", "-------------"
   ELSE
      WRITEFILE "FileB.txt", LineOfText
   ENDIF
ENDWHILE
CLOSEFILE "FileA.txt"
CLOSEFILE "FileB.txt"
```

Two files can be open at the same time, each in its own mode.

exercises.lol exercises.com.cn

10.3.3 Practice

1. A file `Marks.txt` has one whole-number mark per line. Write pseudocode that reads it and outputs the average mark.

<details> <summary>Model answer</summary>

```
DECLARE Line : STRING
DECLARE Total : INTEGER
DECLARE Count : INTEGER
// Create a sample file so that the program can run
OPENFILE "Marks.txt" FOR WRITE
WRITEFILE "Marks.txt", "60"
WRITEFILE "Marks.txt", "75"
WRITEFILE "Marks.txt", "90"
CLOSEFILE "Marks.txt"

Total ← 0
Count ← 0
OPENFILE "Marks.txt" FOR READ
WHILE NOT EOF("Marks.txt")
   READFILE "Marks.txt", Line
   Total ← Total + STR_TO_NUM(Line)
   Count ← Count + 1
ENDWHILE
CLOSEFILE "Marks.txt"
IF Count > 0 THEN
   OUTPUT "Average: ", Total / Count
ELSE
   OUTPUT "The file is empty"
ENDIF
```

Each line is a string, so `STR_TO_NUM` turns it into a number. The `IF Count > 0` stops a division by zero when the file is empty.

</details>

Common mistakes

- Using `REPEAT … UNTIL EOF(…)`. It reads once **before** testing, so it fails on an empty file.
- Using a `FOR` loop with a guessed number of lines.
- Doing arithmetic on the line you read without converting it first.
- Reading a file that is open `FOR WRITE`, or writing to one that is open `FOR READ`.

11 Standard algorithms

11.1 ■ Linear search

A **linear search** checks the elements of an array **one by one**, from the first, until it finds the item or reaches the end. It works on any array, sorted or not.

11.1.1 Stop as soon as it is found

A good search stops early. Use a `WHILE` loop with a **flag** that becomes `TRUE` when the item is found:

```
DECLARE Names : ARRAY[1:6] OF STRING
DECLARE SearchName : STRING
DECLARE Index : INTEGER
DECLARE Found : BOOLEAN
Names[1] ← "Ola"
Names[2] ← "Raj"
Names[3] ← "Kim"
Names[4] ← "Tom"
Names[5] ← "Eva"
Names[6] ← "Ivy"
SearchName ← "Tom"

Found ← FALSE
Index ← 1
WHILE Found = FALSE AND Index <= 6
   IF Names[Index] = SearchName THEN
      Found ← TRUE
   ELSE
      Index ← Index + 1
   ENDIF
ENDWHILE

IF Found THEN
   OUTPUT SearchName, " found at position ", Index
ELSE
   OUTPUT SearchName, " not found"
ENDIF
```

- The loop stops when the item is found **or** there are no more elements.
- `Index` is only increased when the item is **not** found, so after the loop it still points to the item.
- Change `SearchName` to `"Zoe"` and run it again to see the “not found” path.

11.1.2 As a function

In Paper 2 a search is often written as a function that returns the position, or a special value such as -1 when the item is missing:

```
DECLARE Data : ARRAY[1:5] OF INTEGER

FUNCTION FindValue(Target : INTEGER) RETURNS INTEGER
   DECLARE i : INTEGER
   FOR i ← 1 TO 5
      IF Data[i] = Target THEN
         RETURN i
      ENDIF
   NEXT i
   RETURN -1
ENDFUNCTION

Data[1] ← 14
Data[2] ← 3
Data[3] ← 27
Data[4] ← 8
Data[5] ← 19
OUTPUT FindValue(27)
OUTPUT FindValue(5)
```

`RETURN i` leaves the function at once, so the `FOR` loop does not finish. `RETURN -1` is only reached when nothing matched.

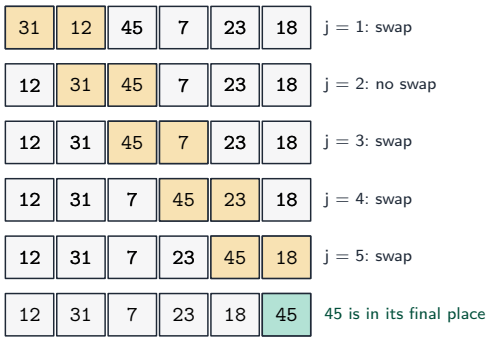
Common mistakes

- Writing `ELSE OUTPUT "Not found"` **inside** the loop. That prints “Not found” for every element that does not match. Decide “not found” only **after** the loop.
- A loop condition that never stops: always include the `Index <= <upper bound>` test.

11.2 ■ Bubble sort

A **bubble sort** puts the elements of an array in order. It goes through the array comparing each **pair of neighbours**, and swaps them if they are in the wrong order. After one full pass pass , the largest value has “bubbled” to the end. Each pass moves one more value into its final place.

exercises.lol exercises.com.cn



One pass of a bubble sort: compare neighbours, swap if out of order

exercises.lol exercises.com.cn

11.2.1 The improved version

The best version stops as soon as a pass makes **no swaps**, because then the array is already sorted. It also checks one fewer element on each pass.

```
DECLARE Numbers : ARRAY[1:6] OF INTEGER
DECLARE Top : INTEGER
DECLARE j : INTEGER
DECLARE Temp : INTEGER
DECLARE Swapped : BOOLEAN
Numbers[1] ← 31
Numbers[2] ← 12
Numbers[3] ← 45
Numbers[4] ← 7
Numbers[5] ← 23
Numbers[6] ← 18

Top ← 5
REPEAT
   Swapped ← FALSE
   FOR j ← 1 TO Top
      IF Numbers[j] > Numbers[j + 1] THEN
         Temp ← Numbers[j]
         Numbers[j] ← Numbers[j + 1]
         Numbers[j + 1] ← Temp
         Swapped ← TRUE
      ENDIF
   NEXT j
   Top ← Top - 1
UNTIL Swapped = FALSE OR Top = 0

FOR j ← 1 TO 6
   OUTPUT Numbers[j]
NEXT j
```

Read it in four parts:

1. **Compare neighbours**: `Numbers[j]` with `Numbers[j + 1]`. The loop goes only to `Top`, so `j + 1` never goes past the end.
2. **Swap** with a temporary variable. Without `Temp`, the first value would be lost.
3. **Remember** that a swap happened: `Swapped ← TRUE`.
4. **Stop** when a pass makes no swaps. `Top ← Top - 1` skips the values already in place.

To sort in **descending** order (largest first), change `>` to `<`.

11.2.2 Sorting records

To sort an array of records, compare one **field**, but swap the **whole records**:

```
IF Students[j].Mark > Students[j + 1].Mark THEN
   Temp ← Students[j]
   Students[j] ← Students[j + 1]
   Students[j + 1] ← Temp
ENDIF
```

Here `Temp` must be declared with the record type.

Common mistakes

- `FOR j ← 1 TO 6` in a 6-element array: then `Numbers[j + 1]` becomes `Numbers[7]`, which does not exist. The inner loop goes to one less than the size.
- Swapping without a temporary variable: `Numbers[j] ← Numbers[j + 1]` then `Numbers[j + 1] ← Numbers[j]` copies the same value into both.
- Setting `Swapped ← FALSE` **outside** the `REPEAT`. It must be reset at the start of every pass.

11.3 ■ Validating input

Validation is an automatic check that input is **sensible** before the program uses it. It cannot prove the data is correct, only that it is reasonable.

Check	Question it asks	Example
range check	Is the value between a lower and upper limit?	a mark from 0 to 100
length check	Does it have an allowed number of characters?	a password of at least 8 characters
type check	Is it the right kind of data?	the age is a number
presence check	Was anything entered at all?	the name is not empty
format check	Does it follow a pattern?	a code of 2 letters then 4 digits

exercises.lol exercises.com.cn

11.3.1 The validation loop

Ask again and again until the input passes. `REPEAT … UNTIL` fits, because you must ask at least once:

```
DECLARE Entry : STRING
DECLARE Mark : INTEGER
DECLARE Valid : BOOLEAN
REPEAT
   OUTPUT "Enter a mark from 0 to 100"
   INPUT Entry
   Valid ← FALSE
   IF IS_NUM(Entry) THEN
      Mark ← STR_TO_NUM(Entry)
      IF Mark >= 0 AND Mark <= 100 THEN
         Valid ← TRUE
      ENDIF
   ENDIF
   IF Valid = FALSE THEN
      OUTPUT "Invalid. Try again."
   ENDIF
UNTIL Valid
OUTPUT "Mark accepted: ", Mark
```

This does a **type check** (`IS_NUM`) first, then a **range check**. The type check must come first, because a range check on text such as `"abc"` makes no sense.

11.3.2 A validation function

When the rules get longer, put them in a function that returns `TRUE` or `FALSE`:

```
FUNCTION ValidPassword(Password : STRING) RETURNS BOOLEAN
   DECLARE i : INTEGER
   DECLARE HasDigit : BOOLEAN
   DECLARE Ch : CHAR
   IF LENGTH(Password) < 8 THEN
      RETURN FALSE
   ENDIF
   HasDigit ← FALSE
   FOR i ← 1 TO LENGTH(Password)
      Ch ← MID(Password, i, 1)
      IF Ch >= '0' AND Ch <= '9' THEN
         HasDigit ← TRUE
      ENDIF
   NEXT i
   RETURN HasDigit
ENDFUNCTION

OUTPUT ValidPassword("sunshine")
OUTPUT ValidPassword("sunshine7")
OUTPUT ValidPassword("ab1")
```

It prints `FALSE` (no digit), `TRUE` and `FALSE` (too short).

11.3.3 Practice

1. Change the bubble sort above so that it sorts the numbers from largest to smallest.

2. Write a function `CountOf(Target : INTEGER) RETURNS INTEGER` that returns how many times `Target` appears in a global array `Data[1:8]`.

<details> <summary>Model answer</summary>

1. Change the comparison to `IF Numbers[j] < Numbers[j + 1] THEN`. Nothing else changes.


```
DECLARE Data : ARRAY[1:8] OF INTEGER

FUNCTION CountOf(Target : INTEGER) RETURNS INTEGER
   DECLARE i : INTEGER
   DECLARE Count : INTEGER
   Count ← 0
   FOR i ← 1 TO 8
      IF Data[i] = Target THEN
         Count ← Count + 1
      ENDIF
   NEXT i
   RETURN Count
ENDFUNCTION

Data[1] ← 4
Data[2] ← 7
Data[3] ← 4
Data[4] ← 1
Data[5] ← 9
Data[6] ← 4
Data[7] ← 2
Data[8] ← 7
OUTPUT CountOf(4)
OUTPUT CountOf(7)
```

It prints `3` and `2`. Here the loop must **not** stop early: every element has to be counted, so a `FOR` loop is right.

</details>

Common mistakes

- Validating with a single `IF`: the user gets one more chance and then the program goes on with bad data. Use a loop.
- Confusing validation with **verification**. Verification checks that data was **copied correctly** (double entry, visual check); validation checks that it is **sensible**.

exercises.lol exercises.com.cn

12 Stacks, queues and linked lists

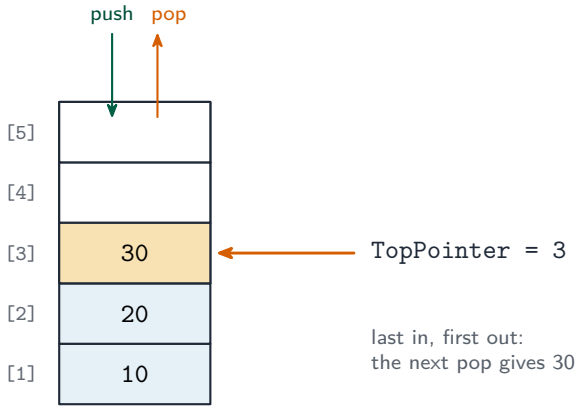
12.1 ■ A stack in an array

An **abstract data type** (ADT) is a collection of data together with the operations allowed on it. The user of an ADT only needs to know **what** the operations do, not how they work inside. At AS you build three ADTs from **arrays**: a stack, a queue and a linked list.

A **stack** is **last in, first out** (LIFO): the last item added is the first item taken off, like a pile of plates.

- **Push** push : add an item to the top.
- **Pop** pop : remove the item at the top.

The array holds the items, and a variable `TopPointer` holds the index of the top item. When the stack is empty, `TopPointer` is 0.



A stack in an array: TopPointer marks the last item pushed


```
CONSTANT MaxSize = 5
DECLARE Stack : ARRAY[1:5] OF INTEGER
DECLARE TopPointer : INTEGER

PROCEDURE Push(Item : INTEGER)
   IF TopPointer = MaxSize THEN
      OUTPUT "Stack full: cannot push ", Item
   ELSE
      TopPointer ← TopPointer + 1
      Stack[TopPointer] ← Item
   ENDIF
ENDPROCEDURE

FUNCTION Pop() RETURNS INTEGER
   DECLARE Item : INTEGER
   IF TopPointer = 0 THEN
      OUTPUT "Stack empty"
      RETURN -1
   ELSE
      Item ← Stack[TopPointer]
      TopPointer ← TopPointer - 1
      RETURN Item
   ENDIF
ENDFUNCTION

TopPointer ← 0
CALL Push(10)
CALL Push(20)
CALL Push(30)
OUTPUT "Popped ", Pop()
OUTPUT "Popped ", Pop()
CALL Push(40)
OUTPUT "Popped ", Pop()
```

The output is 30, 20, then 40: always the **most recent** item that is still there.

- `Push` checks for a **full** stack first; `Pop` checks for an **empty** stack first. Mark schemes look for these checks.
- `Pop` does not need to clear the old value from the array. Moving `TopPointer` down is enough: that space will be written over by the next `Push`.

Common mistakes

- Changing the pointer in the wrong order. In `Push`, increase `TopPointer` **then** store; in `Pop`, read the item **then** decrease `TopPointer`.
- Forgetting the full or empty check.

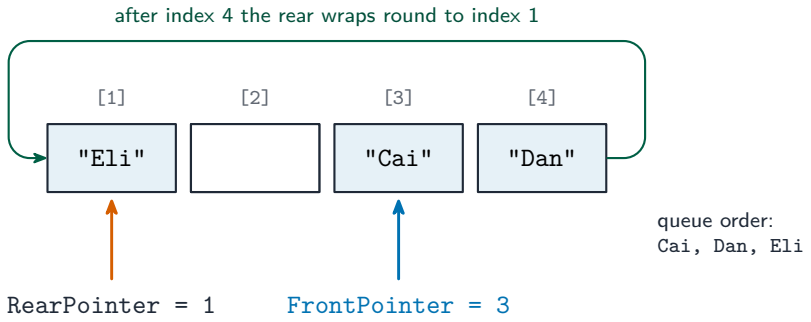
exercises.lol exercises.com.cn

12.2 ■ A circular queue in an array

A **queue** is **first in, first out** (FIFO): items leave in the order they arrived, like people waiting in a line.

- **Enqueue** enqueue : add an item at the **rear**.
- **Dequeue** dequeue : remove the item at the **front**.

If the queue just moved forward in the array, it would soon run off the end even with empty space at the start. A **circular queue** solves this: when a pointer passes the last index, it **wraps round** to index 1 again.



A circular queue: when a pointer passes the last index it goes back to 1

The queue uses three variables:

- `FrontPointer`: the index of the item at the front;
- `RearPointer`: the index of the item at the rear;
- `NumberInQueue`: how many items there are, which makes "full" and "empty" easy to test.

```
CONSTANT MaxSize = 4
DECLARE Queue : ARRAY[1:4] OF STRING
DECLARE FrontPointer : INTEGER
DECLARE RearPointer : INTEGER
DECLARE NumberInQueue : INTEGER

PROCEDURE Enqueue(Item : STRING)
   IF NumberInQueue = MaxSize THEN
      OUTPUT "Queue full"
   ELSE
      RearPointer ← RearPointer + 1
      IF RearPointer > MaxSize THEN
         RearPointer ← 1
      ENDIF
      Queue[RearPointer] ← Item
      NumberInQueue ← NumberInQueue + 1
   ENDIF
ENDPROCEDURE

FUNCTION Dequeue() RETURNS STRING
   DECLARE Item : STRING
   IF NumberInQueue = 0 THEN
      RETURN "EMPTY"
   ELSE
      Item ← Queue[FrontPointer]
      FrontPointer ← FrontPointer + 1
      IF FrontPointer > MaxSize THEN
         FrontPointer ← 1
      ENDIF
      NumberInQueue ← NumberInQueue - 1
      RETURN Item
   ENDIF
ENDFUNCTION

FrontPointer ← 1
RearPointer ← 0
NumberInQueue ← 0
CALL Enqueue("Ann")
CALL Enqueue("Ben")
CALL Enqueue("Cai")
OUTPUT Dequeue()
OUTPUT Dequeue()
CALL Enqueue("Dan")
CALL Enqueue("Eli")
CALL Enqueue("Fay")
OUTPUT Dequeue(), " ", Dequeue(), " ", Dequeue(), " ", Dequeue()
```

`Eli` goes into index 1 again, because `RearPointer` wrapped round. The queue still gives the names back in the order they arrived.

The wrap-round can also be written with `MOD`: `RearPointer ← (RearPointer MOD MaxSize) + 1`. Both forms are fine.

Common mistakes

- Forgetting the wrap-round, so the pointer goes past the last index.
- Starting `RearPointer` at 1. With the version above, it starts at 0, so the first `Enqueue` moves it to 1.

exercises.lol exercises.com.cn

12.3 ■ A linked list in arrays

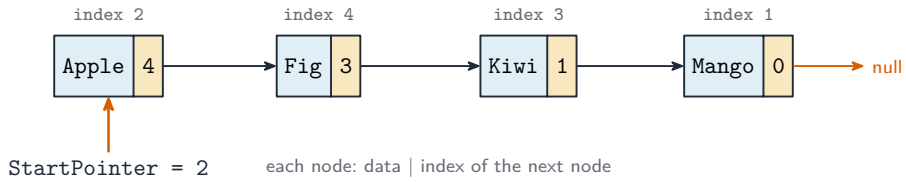
A **linked list** is a list where each item, called a **node**, holds its data **and** a **pointer** to the next node. The items do not have to be stored in order: the pointers give the order.

With arrays, a linked list needs:

- an array for the data, and an array `NextPointer` of the same size;
- `StartPointer`: the index of the **first** node;
- a **null pointer** null pointer, here 0, that marks the end of the list.

Index	Data	NextPointer
1	"Mango"	0
2	"Apple"	4
3	"Kiwi"	1
4	"Fig"	3

With `StartPointer = 2`, the list reads Apple → Fig → Kiwi → Mango, which is alphabetical order, although the array is not.



Following the pointers from StartPointer gives the list order

12.3.1 Following the pointers

To visit every node, start at `StartPointer` and follow each `NextPointer` until you reach the null pointer:

```
CONSTANT NullPointer = 0
DECLARE Data : ARRAY[1:4] OF STRING
DECLARE NextPointer : ARRAY[1:4] OF INTEGER
DECLARE StartPointer : INTEGER
DECLARE Current : INTEGER
Data[1] ← "Mango"
NextPointer[1] ← 0
Data[2] ← "Apple"
NextPointer[2] ← 4
Data[3] ← "Kiwi"
NextPointer[3] ← 1
Data[4] ← "Fig"
NextPointer[4] ← 3
StartPointer ← 2

Current ← StartPointer
WHILE Current <> NullPointer
   OUTPUT Data[Current]
   Current ← NextPointer[Current]
ENDWHILE
```

12.3.2 Adding and removing a node

To add or remove a node, you change **pointers only**; no data moves.

- **Remove** "Kiwi": make the node before it ("Fig", index 4) point to the node after it ("Mango", index 1). Set `NextPointer[4] ← 1`.
- **Add** a new node: store its data in a free element, point it to the node that should follow it, then point the node before it to the new node.

Unused elements are usually kept in a second list, the **free list**, with its own `FreeListPointer`. Full insert and delete algorithms with a free list are A-Level (Paper 4) content; at AS you should be able to follow the pointers and say what changes.

12.3.3 Practice

1. In the table above, which index holds the **last** node of the list? How do you know?

2. Using the queue above, what is the value of `RearPointer` after `"Fay"` is added?

<details> <summary>Model answer</summary>

1. Index 1 (`"Mango"`). Its `NextPointer` is 0, the null pointer, so no node comes after it.

exercises.lol exercises.com.cn

2. `RearPointer` is 2. The six enqueues move it 1, 2, 3, then (after two dequeues) 4, then it wraps round to 1 for `"Eli"` and moves to 2 for `"Fay"`.

</details>

Common mistakes

- Reading the linked list in **index order** (1, 2, 3, 4) instead of following the pointers.
- Mixing up the two ADTs: a stack takes from the **same end** it adds to; a queue takes from the **other** end.
- Writing `Current ← Current + 1` to move along a linked list. The next node is `NextPointer[Current]`, which is usually somewhere else.

exercises.lol exercises.com.cn

13 Quick reference

13.1 ■ Every keyword on one page

Use this page to check a **syntax** form quickly. The topic number tells you where it is explained.

13.1.1 Data and assignment

Form	Example	Topic
`DECLARE <id> : <type>`	`DECLARE Total : INTEGER`	2
`CONSTANT <id> = <value>`	`CONSTANT Rate = 0.08`	2
`<id> ← <value>`	`Total ← Total + 1`	2
`DECLARE <id> : ARRAY[<l>:<u>] OF <type>`	`DECLARE Names : ARRAY[1:30] OF STRING`	6
`DECLARE <id> : ARRAY[<l1>:<u1>, <l2>:<u2>] OF <type>`	`DECLARE Board : ARRAY[1:3, 1:3] OF CHAR`	6
`TYPE <id> … ENDTYPE`	a record with `DECLARE` fields	9

Data types: `INTEGER`, `REAL`, `CHAR`, `STRING`, `BOOLEAN`, `DATE`.

13.1.2 Operators

Kind	Operators
arithmetic	`+ - * / DIV MOD`
relational	`= <> < > <= >=`
logic	`AND OR NOT`
strings	`&`

13.1.3 Input and output

Form	Example
`INPUT <id>`	`INPUT Age`
`OUTPUT <value(s)>`	`OUTPUT "Total: ", Total`

13.1.4 Selection and loops

```
IF <condition> THEN
   <statement(s)>
ELSE
   <statement(s)>
ENDIF

CASE OF <identifier>
   <value> : <statement(s)>
   <value1> TO <value2> : <statement(s)>
   OTHERWISE : <statement(s)>
ENDCASE

FOR <id> ← <v1> TO <v2> STEP <n>
   <statement(s)>
   <statement(s)>
NEXT <id>

WHILE <condition>
   <statement(s)>
ENDWHILE

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

13.1.5 Subroutines

```
PROCEDURE <id>(BYREF <p> : <type>, …)
   <statement(s)>
ENDPROCEDURE

CALL <id>(<arguments>)

FUNCTION <id>(<p> : <type>, …) RETURNS <type>
   <statement(s)>
   RETURN <value>
ENDFUNCTION

<variable> ← <id>(<arguments>)
```

13.1.6 Files

Form	Notes
`OPENFILE <file> FOR READ / WRITE / APPEND`	`WRITE` empties the file
`READFILE <file>, <variable>`	reads the next line
`WRITEFILE <file>, <data>`	writes one line
`EOF(<file>)`	`TRUE` when no lines are left
`CLOSEFILE <file>`	always close

exercises.lol exercises.com.cn

13.1.7 Built-in functions (from the Paper 2 insert)

Function	Returns
`LENGTH(S)`	number of characters
`LEFT(S, x)` / `RIGHT(S, x)`	first / last x characters
`MID(S, x, y)`	y characters from position x
`TO_UPPER(x)` / `TO_LOWER(x)`	x in upper / lower case
`NUM_TO_STR(x)` / `STR_TO_NUM(x)`	number to string / string to number
`IS_NUM(S)`	`TRUE` if S is a valid number
`ASC(c)` / `CHR(x)`	character code / character
`INT(x)`	whole-number part (cuts off, never rounds)
`RAND(x)`	random `REAL` from 0 up to (not including) x

Comments start with `//`. Keywords are in capitals; identifiers start with a letter.

13.2 ■ Coming from IGCSE 0478

Many students learned IGCSE pseudocode first. The two Cambridge syllabuses use **different** rules, and a habit from one is often a mistake in the other. This table shows the 9618 form you must now use.

	IGCSE 0478	AS/A Level 9618
`THEN`	on its own line, under the `IF`	at the end of the `IF` line
constants	`CONSTANT Pi ← 3.142`	`CONSTANT Pi = 3.142`
`DIV` and `MOD`	functions: `DIV(10, 3)`	operators: `10 DIV 3`
joining strings	commas in `OUTPUT` only	the `&` operator
`OTHERWISE` in `CASE`	no colon	with a colon: `OTHERWISE :`
`CASE` values	single values, one statement each	ranges with `TO`, several statements
parameters	always by value	`BYVAL` or `BYREF`
part of a string	`SUBSTRING(S, 2, 3)`	`MID(S, 2, 3)`, also `LEFT` and `RIGHT`
upper case of a string	`UCASE(S)`	`TO_UPPER(S)`
rounding, random	`ROUND(X, 2)`, `RANDOM()`	no `ROUND`; `INT(X)` cuts off; `RAND(x)`
identifiers	start with a capital; letters and digits	start with any letter; underscores allowed

	IGCSE 0478	AS/A Level 9618
data types	five	six: `DATE` is added
records	not used	`TYPE … ENDTYPE`

Here is the same small algorithm in 9618 form. It runs on this page:

```
CONSTANT Rate = 0.5
DECLARE Total : REAL
DECLARE Label : STRING
Total ← 10.0
IF 7 MOD 2 = 1 THEN
   Label ← "Odd, " & "half rate"
   OUTPUT Label, ": ", Total * Rate
ENDIF
```

The IGCSE version would put `THEN` on its own line, write `CONSTANT Rate ← 0.5` and `MOD(7, 2)`, and could not use `&`. The site's IGCSE pseudocode reference shows that form.

13.3 ■ Writing a Paper 2 answer

Paper 2 asks you to write, complete, correct and trace pseudocode. These habits win marks.

13.3.1 Before you write

- **Read the whole question** and underline what must be **output** or **returned**.
- Check whether it asks for a **procedure** or a **function**, and copy the header exactly when it is given: the name, the parameters and the return type.
- Look at the **insert**: it lists every built-in function you may use, with its parameters.

13.3.2 While you write

- **Declare** your variables, one per line, with sensible types.
- **Initialise** totals and counters (`Total ← 0`) before the loop.
- Pick the loop that fits (topic 5): `FOR` for a known count, `WHILE` for "may be zero times", `REPEAT` for "at least once".
- **Indent** every block, and close every block: `ENDIF`, `ENDWHILE`, `NEXT`, `ENDCASE`, `ENDPROCEDURE`, `ENDFUNCTION`.
- Use the **exact** names from the question for arrays, variables and subroutines. If the question calls the array `StockLevel`, do not call it `Stock`.
- Write `CALL` for procedures, and never for functions.
- Use `←` for assignment and `=` only for comparison.

exercises.lol exercises.com.cn

13.3.3 Tracing an algorithm

A **trace table** records the value of each variable after each step, so you can see what an algorithm does.

- Make one column for each variable, and one for the output.
- Write a new value only when it **changes**, going down one row each time.
- Work line by line. Do not guess the result from what you think the algorithm "should" do.

13.3.4 When you finish

- Read your answer as the computer would: does every loop end? Is every variable given a value before it is used?
- Test it in your head with a simple case and with an **edge case** edge case : an empty file, a value exactly on a boundary, a value that is not found.

A **dry run** with a trace table is the best way to find a logic error before the examiner does.