Solutions

Each answer shows the steps, then the **result**.

2.1

(a) `INPUT Name` – or `OUTPUT Name`; a parallelogram is input or output

(b) an assignment: `Total <- Total + 1`

IF ... THEN ... ELSE ... ENDIF

(d) `REPEAT ... UNTIL` – the diamond is **below** the body, so the test comes after it

2.2

(a) **8**

(b) `"ter"`

(c) `"pu"` – positions count from 1, so position 4 is the `p`

(d) **9** – `STR_TO_NUM` gives 9.7 and `INT` takes the whole-number part

2.3

- `Dice <- INT(RAND(6)) + 1`
- `RAND` returns a **real** from 0 up to but not including 6, so `INT` is needed to make it a whole number; the `+ 1` shifts the range from 0-5 to 1-6

2.4

- `+` cannot join a string to an integer: the two are different types
- `Message <- "Total: " & NUM_TO_STR(Score)` – `&` concatenates, and the number must be converted to a string first

3.1

- declarations, with `Total` and `Count` as `INTEGER`
- `Total <- 0` and `Count <- 0` **before** the loop
- a `REPEAT ... UNTIL Temperature = -99` loop containing the `INPUT`
- adding to the total and incrementing the count **only** when the value is not −99
- the two `OUTPUT` statements after the loop


```
DECLARE Temperature, Total, Count : INTEGER
Total <- 0
Count <- 0
REPEAT
    OUTPUT "Enter a temperature:"
    INPUT Temperature
    IF Temperature <> -99 THEN
        Total <- Total + Temperature
        Count <- Count + 1
    ENDIF
UNTIL Temperature = -99
OUTPUT "Total: ", Total
OUTPUT "Number entered: ", Count
```

- the `IF` inside the loop is the mark most often dropped: without it the sentinel −99 is added to the total and counted

3.2

- `Count <- 1` before the loop
- a `REPEAT ... UNTIL` loop, because the diamond is **below** the body
- `INPUT Name` and `OUTPUT Name` inside it
- `Count <- Count + 1` inside it
- `UNTIL Count > 5` – the **negation** of the diamond's `Count <= 5?`, whose `Yes` loops back

```
DECLARE Count : INTEGER
DECLARE Name : STRING
Count <- 1
REPEAT
    INPUT Name
    OUTPUT Name
    Count <- Count + 1
UNTIL Count > 5
```

- a `WHILE` version scores nothing for the loop mark: the chart tests after the body, so the body must always run at least once

3.3

(a) the inner call runs first: `RIGHT("Program", 2)` is `"am"`, so the answer is `"AM"`

(b) `MID("Computer", 3, 4)` is `"mput"`, so `LENGTH` of it is **4**

(c) `'Q'` is 81 and `'A'` is 65, so the answer is **16**

3.4

- `LEFT(Surname, 3)` for the first three letters
- `TO_UPPER(...)` around it: it accepts a whole string, whereas `UCASE` takes one `CHAR` and is not on the insert
- `LEFT(FirstName, 1)` for the initial

- `NUM_TO_STR(Year)`, and the three parts joined with `&`

```
Username <- TO_UPPER(LEFT(Surname, 3)) & LEFT(FirstName, 1) &
↪  NUM_TO_STR(Year)
```

- for `Surname` `"Chen"`, `FirstName` `"Yuwei"` and `Year` 12 this gives `"CHEY12"`

3.5 any three:

- the routines have already been written, compiled and **tested**, so they are less likely to contain errors
- they save development time, because the code does not have to be written again
- they may do things the programmer could not write themselves, such as complex statistics or graphics; they are written by experts; and a routine with a fixed interface can be called from anywhere in the program, and reused across many programs