A-LEVEL COMPUTER SCIENCE • EXERCISE SHEET

16.2.1 Recursive BNF, syntax diagrams, and RPN in both directions

Name: _____ Class: _____ Date: _____ **Total: 27 marks**

KEY WORDS syntax diagram 语法图 • backus-naur form 巴科斯-诺尔范式 • reverse polish notation 逆波兰表示法 • machine code 机器码 • infix 中缀

Objective

Build the skills to answer the **translation software** questions at Paper 3 depth. Sheet 16.2 names the compilation stages, writes a one-line **BNF** rule and converts a short expression to **RPN** 逆波兰表示法. This sheet is the same four statements where the marks actually are: **recursive** BNF, which is how a grammar describes something of any length; **syntax diagrams** 语法图 as the same grammar drawn rather than written; and RPN in **both** directions, traced on a **stack** 栈.

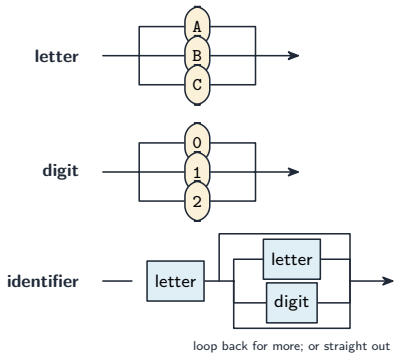
You must be able to:

- show understanding of how the grammar of a language can be expressed using **syntax diagrams** or **Backus-Naur Form** (BNF) notation, including **recursive** rules
- convert between a syntax diagram and a BNF rule
- show understanding of how RPN can be used to evaluate expressions, converting **infix to RPN** and **RPN back to infix**
- evaluate an RPN expression using a stack, showing the stack at each step

1 Worked examples

Study these first. Each one shows a **method** that a question later on this sheet needs.

■ BNF, and the recursion that does the real work



the same rules in BNF

```
<letter> ::= A | B | C
<digit> ::= 0 | 1 | 2
<tail> ::= <letter> | <digit> | <letter><tail> | <digit><tail>
<identifier> ::= <letter> | <letter><tail>
```

a loop in the diagram becomes a rule that refers to itself (recursion); the first symbol is fixed by the first box on the rail

valid: `A`, `AB2`, `C01B` invalid: `2A` (starts with a digit), `AD` (D is not a letter here)

A BNF rule defines one **non-terminal** in terms of terminals and other non-terminals:

```
<digit> ::= 0|1|2|3|4|5|6|7|8|9
```

That alone can only ever describe **one** character. To describe a number of **any length**, the rule must refer to **itself**:

```
<integer> ::= <digit> | <digit><integer>
```

Read it as: an integer is either a single digit, **or** a digit followed by another integer. That second alternative is the **recursion**, and it is what makes the rule unbounded.

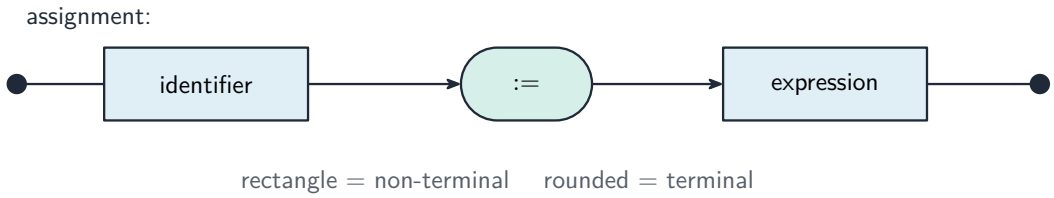
- **Every recursive rule needs a non-recursive alternative** – here `<digit>` – or nothing can ever terminate. A rule written as `<integer> ::= <digit><integer>` alone defines nothing at all, and that is the error the mark scheme looks for.
- The recursive part goes on **one side**. `<digit><integer>` builds left to right; `<integer><digit>` builds right to left. Both describe the same set here, but mixing them in one alternative does not.

Worked example. *Write BNF for an identifier that starts with a letter and then contains any number of letters or digits.*

```
<letter>     ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit>      ::= 0|1|2|3|4|5|6|7|8|9
<identifier> ::= <letter> | <identifier><letter> | <identifier><digit>
```

The base case is a single `<letter>`, which is what forces the first character to be a letter; the two recursive alternatives then add one character at a time to the right.

■ **A syntax diagram is the same grammar, drawn**



The translation between the two notations is mechanical:

In BNF	In a syntax diagram
a **terminal**, such as `7`	a **rounded** box (or circle)
a **non-terminal**, such as `<digit>`	a **rectangular** box
‘	‘, an alternative
one item after another	boxes in sequence along the line
recursion	a line that **loops back**

So a loop in a diagram is a recursive rule in BNF, and a branch is a `|`. A question giving you one and asking for the other is asking you to apply that table.

■ RPN: why it exists, and both directions

In **Reverse Polish Notation** the operator comes **after** its two operands. It needs **no brackets** and no rules of precedence, because the order of the symbols already fixes the order of the operations – which is exactly why a compiler generates it.

Infix to RPN. Bracket the expression fully according to precedence, then move each operator to just after its own pair of operands.

Convert $(3 + 4) \times 2$.

The bracket is done first, so 3+4 becomes `3 4 +`. That whole result is then multiplied by 2:

```
3 4 + 2 *
```

Convert $A + B \times C$. Multiplication binds tighter, so it is $A + (B \times C)$, giving `A B C * +` – **not** `A B + C *`, which is $(A + B) \times C$.

RPN to infix. Work left to right; every time an operator appears, take the **two most recent** items, write them with the operator between them in brackets, and put that back.

Convert `4 5 + 2 6 - *`. → `(4 + 5)`, then `(2 - 6)`, then `((4 + 5) * (2 - 6))`.

■ Evaluating with a stack

Read left to right. A **number is pushed**; an **operator pops two**, applies itself and pushes the result. The order matters: the **first** value popped is the **right-hand** operand.

Worked example. *Evaluate* `5 2 - 3 *`.

Token	Action	Stack after
`5`	push 5	5
`2`	push 2	5, 2
`-`	pop 2 then 5; $5 - 2 = 3$; push	3
`3`	push 3	3, 3
`*`	pop 3 then 3; $3 \times 3 = 9$; push	9

The answer is the single value left on the stack: **9**.

- For `-` and `/` the order is the whole question. Popping gives the **second** operand first, so it is second popped − first popped. Getting it backwards turns $5 - 2$ into $2 - 5$.

2 Practice

Now use the methods above.

2.1 Write a BNF rule for `<integer>`, a whole number of one or more digits. You may use `<digit>` without defining it. [2]

2.2 State why a recursive BNF rule must also have a **non-recursive** alternative. [1]

2.3 In a syntax diagram, state what is shown by a **rounded** box, by a **rectangular** box, and by a line that **loops back**. [3]

2.4 Convert the infix expression $A + B \times C$ to RPN. [2]

2.5 Convert the RPN expression `7 2 3 * -` back to infix, and give its value. [2]

3 Exam-style questions

3.1 A programming language defines a variable name as starting with a letter, followed by any number of letters or digits.

Write the BNF rules for `<letter>`, `<digit>` and `<identifier>`. [4]

3.2 A student writes the rule `<integer> ::= <digit><integer>`.

Explain why this rule defines no valid integer, and correct it. [3]

3.3 Convert the infix expression $(8 - 3) + (4/2)$ to Reverse Polish Notation. [3]

3.4 Evaluate the RPN expression `4 5 + 2 6 - *` using a stack.

Show the contents of the stack after each token is processed. [5]

3.5 State **two** reasons why a compiler converts expressions into Reverse Polish Notation. [2]

Solutions

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

2.1

- `<integer> ::= <digit> | <digit><integer>`
- one mark for the single-digit base case, one for the recursive alternative; `<integer><digit>` is equally acceptable

2.2

- without a non-recursive alternative the rule refers to itself for ever and never reaches a terminal, so no string can be derived from it

2.3

- a **rounded** box is a **terminal** – a literal symbol that appears in the language itself
- a **rectangular** box is a **non-terminal** – a name defined by another rule
- a line that **loops back** is **repetition**, which in BNF is written as recursion

2.4

- multiplication binds more tightly, so the expression is $A + (B \times C)$
- `A B C * +`
- `A B + C *` is $(A + B) \times C$ and is a different expression

2.5

- the `*` takes 2 and 3, and the `-` then takes 7 and that result: $7 - (2 \times 3)$
- $= \mathbf{1}$

3.1

- `<letter> ::= a|b|c| ... |z`
- `<digit> ::= 0|1|2|3|4|5|6|7|8|9`
- `<identifier> ::= <letter>` as the base case, which forces the first character to be a letter
- `| <identifier><letter> | <identifier><digit>` as the recursive alternatives

```
<letter>     ::= a|b|c|d|e|f|g|h|i|j|k|l|m|n|o|p|q|r|s|t|u|v|w|x|y|z
<digit>      ::= 0|1|2|3|4|5|6|7|8|9
<identifier> ::= <letter> | <identifier><letter> | <identifier><digit>
```

- a rule whose base case is `<letter>|<digit>` allows an identifier to start with a digit, and loses the third mark

3.2

- the rule is **entirely recursive**: every alternative contains `<integer>` again
- so a derivation can never terminate – there is no way to reach a string made only of terminals

- add a non-recursive alternative: `<integer> ::= <digit> | <digit><integer>`

3.3

- each bracket is converted first: $8 - 3$ gives `8 3 -`, and $4/2$ gives `4 2 /`
- the `+` acts on those two results, so it comes last
- `8 3 - 4 2 / +`
- the value is $5 + 2 = 7$, which checks

3.4

- `4` and `5` pushed, then `+` pops both and pushes 9
- `2` and `6` pushed onto the 9
- `-` pops 6 then 2, giving $2 - 6 = -4$, and pushes it
- `*` pops -4 then 9, giving $9 \times (-4)$
- the stack is left holding $-$**36**

Token	Stack after
`4`	4
`5`	4, 5
`+`	9
`2`	9, 2
`6`	9, 2, 6
`-`	9, -4
`*`	-36

- popping in the wrong order gives $6 - 2 = 4$ and a final answer of 36, with the sign lost

3.5 any two:

- RPN needs **no brackets**, so the expression can be evaluated in a single left-to-right pass
- the **order of operations is fixed by the order of the symbols**, so no rules of precedence have to be applied while evaluating
- it is evaluated with a simple **stack**, which is straightforward to implement in machine code; and each operator is met only when both its operands are already available