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.

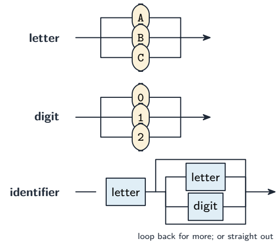
You must be able to:

- 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.

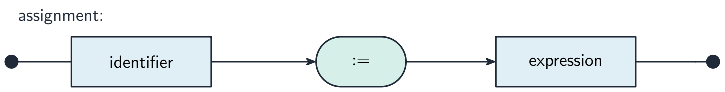
- **Every recursive rule needs a non-recursive alternative** – here <digit> – or nothing can ever terminate. A rule written as <integer> ::= <digit><integer> alone defines nothing at all, and that is the error the mark scheme looks for.
- The recursive part goes on **one side**. <digit><integer> builds 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



rectangle = non-terminal rounded = terminal

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]