

Exercise walk-through

16.2.1

Recursive BNF, syntax diagrams, and RPN in both directions

A-Level Computer Science

BNF – a grammar in text

Backus-Naur Form – writes a
grammar as production rules: `<digit> ::=`

`0 | 1 | ... | 9`

`<identifier> ::= <letter>`

`| <identifier> <letter>`

`| <identifier> <digit> < >` are

non-terminals ; literals are

terminals . Recursion expresses “any
 number of”.

A **syntax diagram**

(railroad diagram) shows the
 same grammar graphically.

The parser uses either to de-
 cide whether a program is
 valid.

Reverse Polish Notation needs no brackets

Infix: operator between operands, so you need brackets and precedence.

RPN

(postfix): operator *follows* its operands.

$(3 + 4) \times 2$



3 4 + 2 ×

Scan left to right with an operator **stack** : output operands; pop higher-or-equal precedence operators before pushing; parentheses flush the stack.

Worked example – evaluate RPN

Evaluate 3 4 2 * +

Push each operand; on an operator, pop two, apply, push the result.

Token	Stack
-------	-------

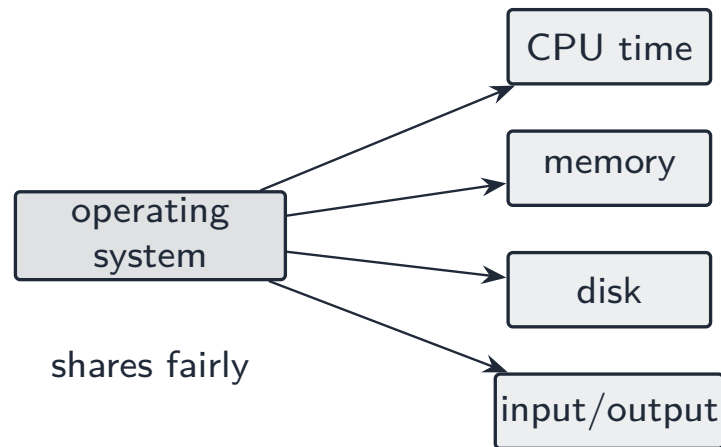
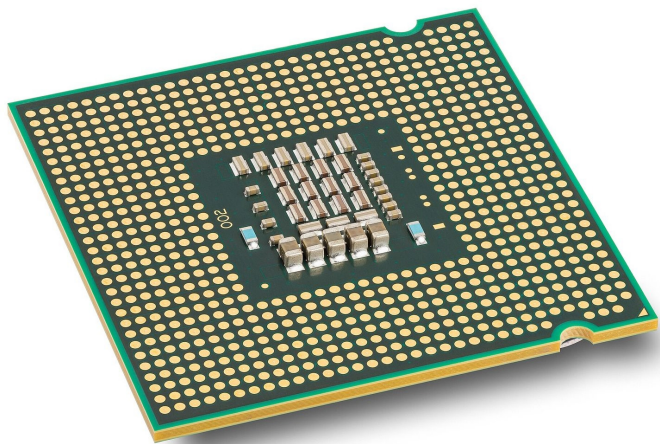
3	3
4	3, 4
2	3, 4, 2
*	3, 8
+	11

$$(A + B) \times (C - D)$$

Convert: A B + C D - ×

Evaluate $(3 + 4) \times (5 - 2)$: push 3, 4; + → 7; push 5, 2; - → 3; × → 21.

How an OS maximises use of resources



*The OS shares the CPU, memory, disk and I/O
between programs*

The processor is a key resource the OS shares between competing tasks

What a scheduler is trading off

responsiveness

how quickly an interactive process gets a turn

throughput

how many processes complete per unit time

fairness

every process getting a share, and none starving

the trade-off

you cannot maximise all three – and each algorithm picks a different corner

first come, first served

maximum fairness by arrival, and terrible responsiveness behind a long job

round robin

good responsiveness; a short time slice costs throughput in context switching

Naming the trade-off is what turns a list of algorithms into an answer. Say which of the three a given algorithm sacrifices.

Exercises 2.1·2.2

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]

Exercises 2.1-2.2

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

2.1

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

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

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

Exercises 2.3-2.4

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]

Exercises 2.3-2.4

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]

2.3

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

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

Exercise 2.5

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

Exercise 2.5

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

2.5

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

Exercise 3.1

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]

Exercise 3.1

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]

3.1

- `<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

Exercises 3.2-3.3

3.2 A student writes the rule $\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \langle \text{integer} \rangle$.

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]

Exercises 3.2-3.3

3.2 A student writes the rule $\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \langle \text{integer} \rangle$.

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

3.2

- the rule is **entirely recursive**: every alternative contains $\langle \text{integer} \rangle$ again
- so a derivation can never terminate – there is no way to reach a string made only of terminals
- add a non-recursive alternative: $\langle \text{integer} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{integer} \rangle$

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

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

Exercise 3.4

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]

Exercise 3.4

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]

3.4

- 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

Exercise 3.5

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

Exercise 3.5

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

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