

System Software

A-Level Computer Science 9618 · Topic 16 revision notes

Everything from the 5 lessons of this topic on a few pages: the key idea of each lesson, the diagram to remember, the points to learn and the traps examiners set. Read it, cover it, test yourself.

1. Process states and scheduling

3. The stages of compilation

2. Paging, segmentation and virtual memory.

4. BNF and syntax diagrams

5. Reverse Polish notation

1 Process states and scheduling

Syllabus 16.1

A **process** is a program that has been loaded into memory and is being executed. In **multi-tasking** the OS switches the processor between processes so quickly that they appear to run at the same time. Each process is **running** (on the processor), **ready** (waiting only for the processor) or **blocked** (waiting for an event such as I/O). The **scheduler** chooses the next ready process using a routine such as **first come first served**, **shortest job first**, **shortest remaining time** or **round robin**. The **kernel** acts as the **interrupt handler**: timer and I/O interrupts trigger the **low-level scheduler** to change which process runs.

The three process states and the transitions between them

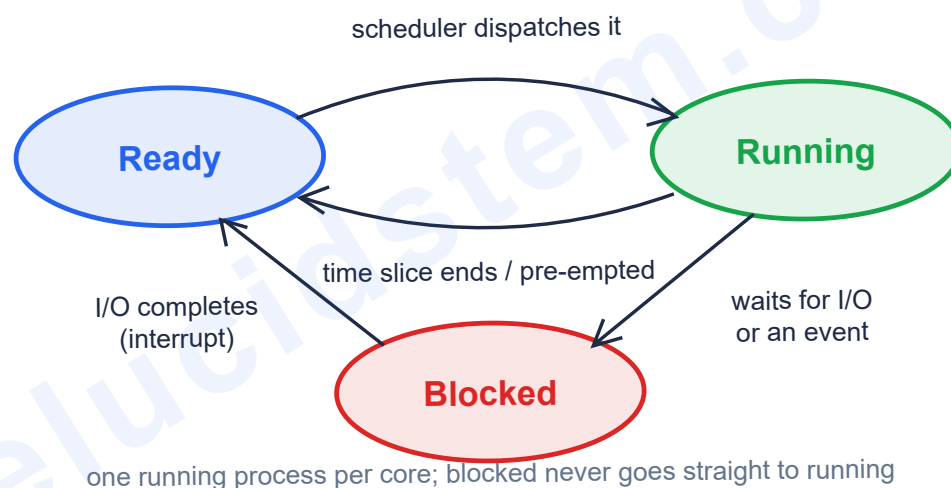


Fig. 1: the running, ready and blocked states and the events that move a process between them

Worked example: Four jobs, three routines

Jobs A, B, C and D all arrive at time 0 in that order, needing 6, 2, 8 and 3 ms of processor time. Find the order of execution and the average waiting time for FCFS, SJF and round robin with a 4 ms time slice.

1. Waiting time = finish time - arrival time - burst time.
2. FCFS: A 0-6, B 6-8, C 8-16, D 16-19. Waits: A 0, B 6, C 8, D 16. Average = $30 / 4 = 7.5$ ms.
3. SJF: shortest first, so B 0-2, D 2-5, A 5-11, C 11-19. Waits: B 0, D 2, A 5, C 11. Average = $18 / 4 = 4.5$ ms.

4. Round robin, 4 ms slice: A 0-4 (2 left), B 4-6 (done), C 6-10 (4 left), D 10-13 (done), A 13-15 (done), C 15-19 (done).

5. RR waits: A $15 - 6 = 9$, B $6 - 2 = 4$, C $19 - 8 = 11$, D $13 - 3 = 10$. Average = $34 / 4 = 8.5$ ms.

Answer: FCFS 7.5 ms, SJF 4.5 ms, RR 8.5 ms. SJF minimises the average wait; RR gives every job a turn early (all four have started by 10 ms) at the cost of a higher average.

State, routine, trigger.

- ✓ **States:** running, ready (needs only the CPU), blocked (needs an event); blocked goes to ready, never straight to running.
- ✓ **Non-pre-emptive:** FCFS (arrival order) and SJF (shortest burst).
- ✓ **Pre-emptive:** SRT (least time left; a new shorter job interrupts) and RR (fixed time slice, back of the queue).
- ✓ **Waiting time** = finish - arrival - burst. Draw the timeline first, then calculate.

Watch out!

A blocked process never goes straight to running. When its I/O completes it joins the **ready** queue and waits to be scheduled like everyone else.

SJF is **non-pre-emptive**: once a job starts it runs to the end. If a question lets a newly arrived short job interrupt the running one, it is describing **SRT**.

2 Paging, segmentation and virtual memory

Syllabus 16.1

In **paging**, a process is divided into fixed-size **pages** and RAM into **page frames** of the same size; a **page table** maps each page to a frame, so a process need not be stored contiguously. In **segmentation**, a process is divided into variable-size logical **segments** (e.g. code, data, stack) recorded in a **segment table**. **Virtual memory** uses secondary storage as if it were RAM: pages not currently needed are kept on disk and **swapped** in when used, causing a **page fault**. A **page replacement** algorithm chooses which page to swap out. If pages are swapped in and out so often that little useful processing is done, the system suffers **disk thrashing**.

Paging: a page table maps each page to a frame in RAM or to disk

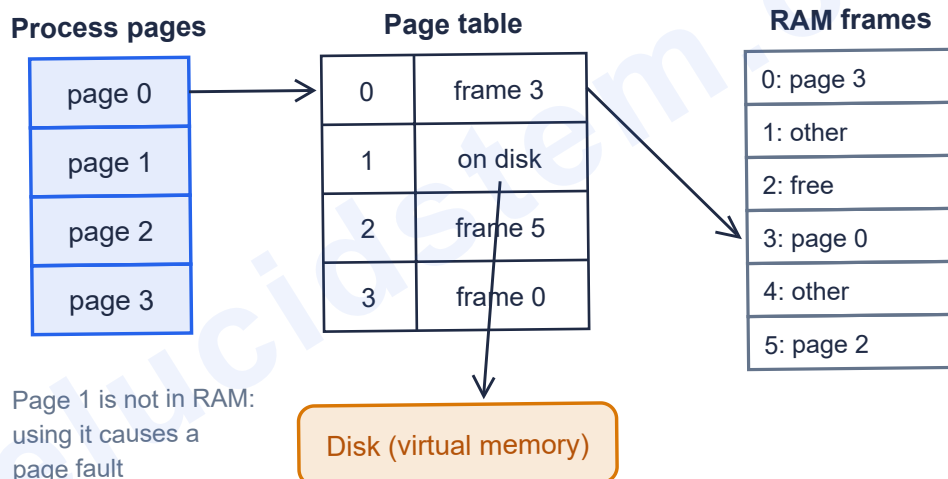


Fig. 1: a page table maps each page of a process to a frame in RAM, or records that the page is on disk

Worked example: Logical to physical address

Pages are 4096 bytes. A process uses logical address 9300. Using the page table in Fig. 1 (page 2 is in frame 5), find the physical address.

1. Page number = $9300 \text{ DIV } 4096 = 2$ (because $2 \times 4096 = 8192$ and $3 \times 4096 = 12288$ is too big).
2. Offset = $9300 \text{ MOD } 4096 = 9300 - 8192 = 1108$.
3. Look up page 2 in the page table: it is in frame 5.
4. Physical address = frame $\times$ page size + offset = $5 \times 4096 + 1108 = 20480 + 1108 = 21588$.

Answer: Physical address 21588. The offset is unchanged; only the page number is replaced by the frame number.

Pages, segments, swaps.

- ✓ **Paging:** fixed-size pages into frames; page table; page number + offset.
- ✓ **Segmentation:** variable-size logical segments; segment table with base and length.
- ✓ **Virtual memory:** disk used as RAM; page fault loads a page; thrashing when swapping dominates.
- ✓ **Replacement:** FIFO = oldest loaded; LRU = longest since last use. Trace frame by frame and count faults.

Watch out!

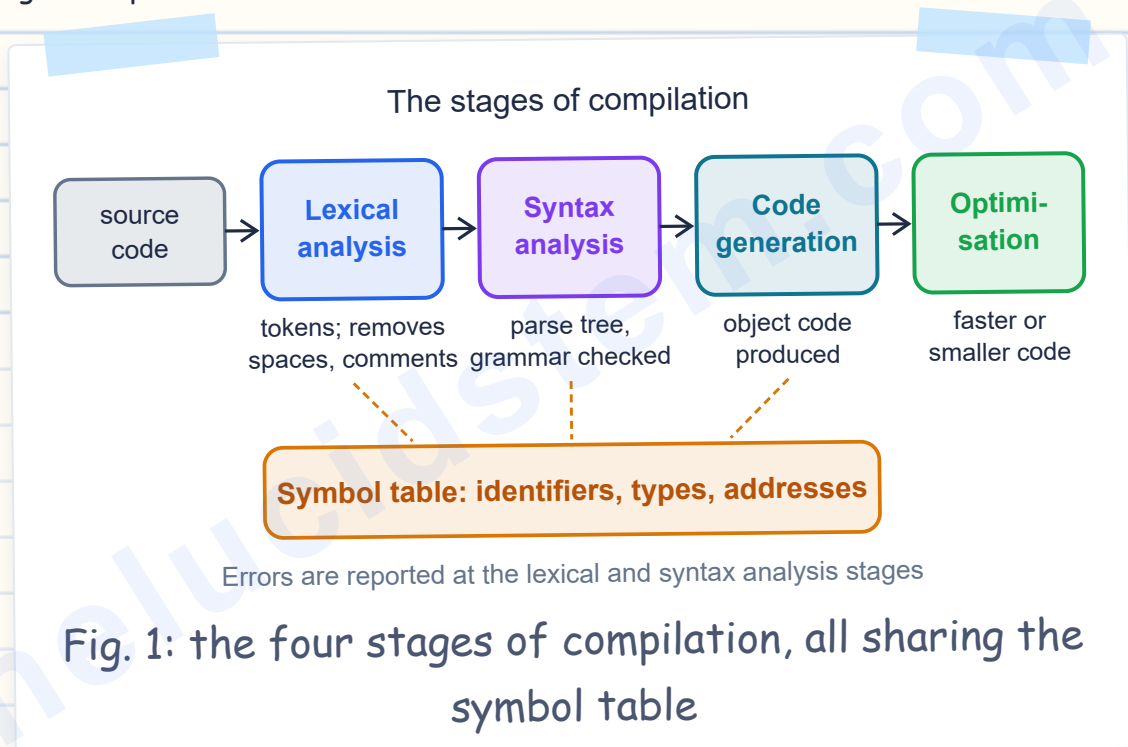
Do not say segments and pages are both fixed size. Pages are **fixed** size and a physical division; segments are **variable** size and a logical division.

Thrashing is not just "using virtual memory". It is when **swapping becomes so frequent** that the processor spends most of its time waiting for pages, so little useful work is done.

3 The stages of compilation

Syllabus 16.2

Compilation has four main stages. **Lexical analysis** removes white space and comments and converts the source code into **tokens**, adding identifiers and constants to the **symbol table**. **Syntax analysis** (parsing) checks that the sequence of tokens obeys the **grammar** of the language, builds a **parse tree** and reports syntax errors. **Code generation** produces **object code** from the parse tree. **Optimisation** makes the object code run faster or use less memory without changing what it does. An **interpreter** analyses each statement in turn and executes it by calling its own pre-written machine code routines, so **no translated version** of the program is produced.



Worked example: Tokenise a statement

Show what lexical analysis produces from this line of source code:

```
Total ← Total + Price * 2 // add this item
```

1. Remove the comment `// add this item` and the extra spaces: they are not needed to execute the program.
2. Split the remaining characters into tokens: `Total`, `←`, `Total`, `+`, `Price`, `*`, `2`. That is 7 tokens.
3. Classify each token: `Total` and `Price` are identifiers, `←` `+` and `*` are operators, `2` is a numeric constant.

4. Add Total, Price and the constant 2 to the symbol table (once each) so later stages can record their data type and address, and replace each identifier with a reference to its symbol table entry.

Answer: A token stream: identifier(Total) assign identifier(Total) plus identifier(Price) multiply constant(2), with Total, Price and 2 in the symbol table.

Four stages, one table.

- ✓ **Lexical analysis:** remove spaces and comments, tokens, keyword table, symbol table.
- ✓ **Syntax analysis:** check tokens against the grammar, parse tree, syntax errors listed.
- ✓ **Code generation:** object code, addresses from the symbol table.
- ✓ **Optimisation:** faster and/or smaller code, same result. **Interpreter:** analyse then execute with its own routines; nothing saved.

Watch out!

Saying an interpreter "translates line by line into machine code" loses the mark at A Level. The point is that it **executes** each statement using its own routines **without producing** a translated version.

Lexical analysis does **not** check the order of the tokens. `Total`
`+ ← Price` passes lexical analysis (every token is valid) and is rejected at **syntax** analysis.

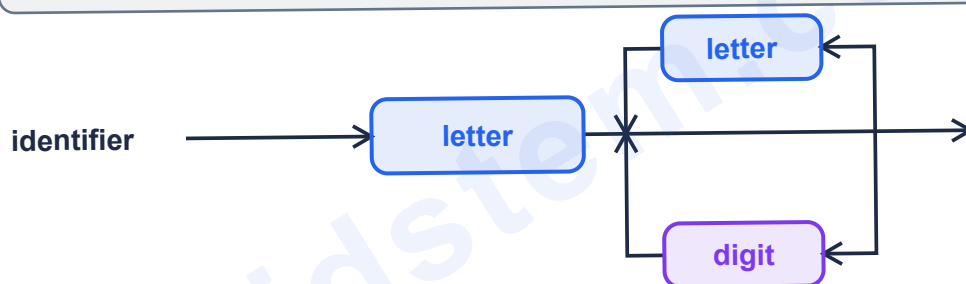
4 BNF and syntax diagrams

Syllabus 16.2

Backus-Naur Form (BNF) is a formal notation for the **grammar** (syntax rules) of a language. Each rule has a **non-terminal** in angle brackets on the left, **::=** meaning "is defined as", and on the right a sequence of terminals and non-terminals, with **|** separating alternatives. **Terminals** are the actual characters or symbols that appear in the language. Repetition is expressed by **recursion**: a rule that refers to itself. A **syntax diagram** shows the same rules graphically, following arrows through boxes, with loops for repetition.

The same rule as BNF and as a syntax diagram

```
<identifier> ::= <letter> | <identifier><letter>  
| <identifier><digit>
```



Start with one letter, then loop back for any number of letters or digits

Fig. 1: one recursive BNF rule and the equivalent syntax diagram

Worked example: Valid or not?

Using the grammar above, decide whether BAD5, 3AB and AF are valid identifiers. Show a derivation for any valid one.

1. BAD5 ends in a digit, so use $\langle \text{identifier} \rangle ::= \langle \text{identifier} \rangle \langle \text{digit} \rangle$: $\langle \text{identifier} \rangle \rightarrow \langle \text{identifier} \rangle \langle \text{digit} \rangle$.
2. Expand the inner identifier twice with $\langle \text{identifier} \rangle \langle \text{letter} \rangle$: $\rightarrow \langle \text{identifier} \rangle \langle \text{letter} \rangle \langle \text{digit} \rangle \rightarrow \langle \text{identifier} \rangle \langle \text{letter} \rangle \langle \text{letter} \rangle \langle \text{digit} \rangle$.
3. End the recursion with $\langle \text{identifier} \rangle ::= \langle \text{letter} \rangle$: $\rightarrow \langle \text{letter} \rangle \langle \text{letter} \rangle \langle \text{letter} \rangle \langle \text{digit} \rangle \rightarrow \text{B A D 5}$. All four terminals are allowed, so BAD5 is valid.
4. 3AB: every identifier must begin with $\langle \text{letter} \rangle$, and 3 is a digit. Invalid.
5. AF: F is not one of the terminals A to E in $\langle \text{letter} \rangle$. Invalid.

Answer: BAD5 valid; 3AB invalid (starts with a digit); AF invalid (F is not a defined letter).

Smallest parts first.

- ✓ **Notation:** $\langle \text{non-terminal} \rangle$, $::=$ is defined as, | or, terminals written as themselves.
- ✓ **Recursion:** "one or more" means a rule that refers to itself plus a non-recursive way out.
- ✓ **Syntax diagrams:** rectangles for non-terminals, rounded boxes for terminals, branches for |, loops for repetition.
- ✓ **Validity:** check every character against the listed terminals and the required order and length.

Watch out!

Check every character against the terminals that are **actually listed**. In exam grammars $\langle \text{letter} \rangle$ or $\langle \text{digit} \rangle$ is often deliberately restricted, e.g. to A to E or to 1 to 9.

A rule like $\langle \text{number} \rangle ::= \langle \text{digit} \rangle \langle \text{number} \rangle$ with **no** non-recursive alternative can never finish, so it defines no strings at all.

Every recursive rule needs a way out.

5 Reverse Polish notation

Syllabus 16.2

In **infix** notation the operator is written between its operands ($A + B$). In **Reverse Polish notation (RPN)**, also called **postfix**, the operator is written after its operands ($A B +$). RPN needs **no brackets** and **no precedence rules**: the order of the tokens fixes the order of the operations. It is evaluated left to right using a **stack**: **push** each operand; for each operator, **pop** two values, apply the operator and push the result. Compilers and interpreters convert expressions to RPN because this simple, unambiguous process is easy to turn into machine code.

$(A + B) * (C - D)$ as a tree, and in Reverse Polish notation

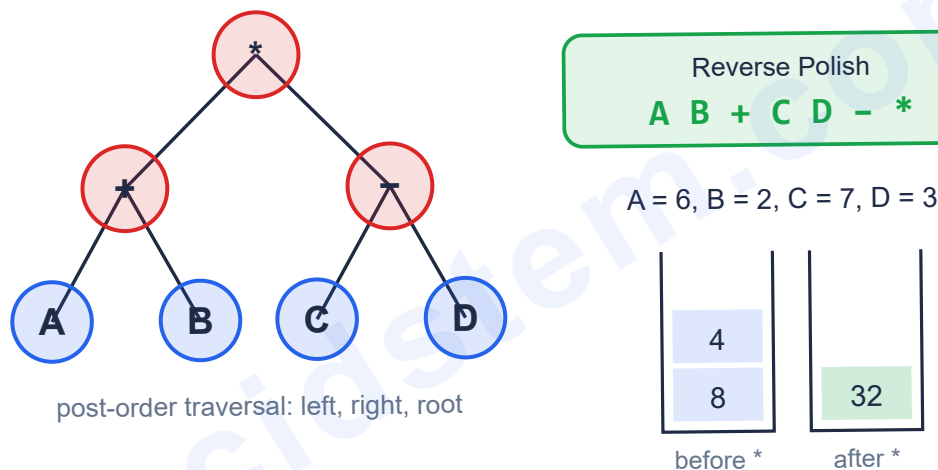


Fig. 1: the expression tree for $(A + B) * (C - D)$, its RPN form from a post-order traversal, and the stack before and after the final *

Worked example: Show the stack contents

Evaluate $A B + C D - *$ when $A = 6$, $B = 2$, $C = 7$ and $D = 3$, showing the contents of the stack after each token.

1. A: push 6. Stack: 6.
2. B: push 2. Stack: 6, 2.
3. +: pop 2 and 6, $6 + 2 = 8$, push. Stack: 8.
4. C: push 7. Stack: 8, 7. D: push 3. Stack: 8, 7, 3.
5. -: pop 3 and 7, $7 - 3 = 4$, push. Stack: 8, 4.
6. *: pop 4 and 8, $8 * 4 = 32$, push. Stack: 32. End of expression.

Answer: 32. Check with infix: $(6 + 2) * (7 - 3) = 8 * 4 = 32$.

Operands push, operators pop.

- ✓ **RPN:** operator after its two operands; operands stay in the same order as in infix.
- ✓ **Converting:** brackets first, then * and /, then + and -; equal precedence left to right.
- ✓ **Evaluating:** push operands; on an operator pop right, pop left, push left op right.
- ✓ **Show the stack** after every token: that is where the marks are.

Watch out!

$A - B - C$ means $(A - B) - C$, so its RPN is $A B - C -$. Writing $A B C - -$ calculates $A - (B - C)$, which gives a different answer.

With - and /, order matters.

The **first** value popped is the **right-hand** operand. For $11\ 4 -$ the answer is $11 - 4 = 7$, not $4 - 11$.