

Cambridge AS & A Level · Computer Science 9618 · Notes

System software A2

Read online at pastopic.com/cambridge/computer-science-9618/notes/system-software-a-level

Last checked 30 September 2026

Read first: [System software](#), [Processor fundamentals](#)

© 2026 Pastopic. Free for personal study. Please share the link, not the file.

What you need to know

This is the A Level topic (Paper 3). What an OS is for, utility software, program libraries and the basic jobs of compilers, interpreters and assemblers are in the AS topic **System software**; interrupts and the stack start in **Processor fundamentals**. By the end of this topic you should be able to:

Purposes of an operating system

1. **Explain how an OS makes the best use of the computer's resources** (processor, memory and devices).
2. **Describe how the user interface hides the complexity of the hardware** from the user.
3. **Explain process management**: what a process is and what multi-tasking means; the three process states (running, ready and blocked); why scheduling is needed; how the four scheduling routines work and what each is good for (round robin, shortest job first, first come first served and shortest remaining time); how the **kernel** acts as an **interrupt handler**, and how interrupt handling is used for **low-level scheduling**.
4. **Explain memory management**: what virtual memory, paging and segmentation are; the difference between paging and segmentation; how pages are replaced; and how **disk thrashing** happens.

Translation software

5. **Explain how an interpreter runs a program without producing a translated version** of it.
6. **Explain the stages of compilation**: lexical analysis, syntax analysis, code generation and optimisation.
7. **Show how the grammar of a language is written** as syntax diagrams or in Backus-Naur Form (BNF).
8. **Show how Reverse Polish Notation (RPN) is used to evaluate expressions**, including converting between infix and RPN and evaluating RPN with a stack.

Understand it

Making the best use of resources

A computer has one processor (or a few cores), a limited amount of main memory (RAM) and slow devices such as disks and printers. The OS shares these out so that as little as possible sits idle:

- **Processor:** while one program waits for a slow device, the OS gives the processor to another, so it is kept busy. The **scheduler** decides who runs next.
- **Memory:** the OS decides which programs, or which parts of them, are in RAM, keeps them apart, and uses **virtual memory** so that more programs fit than the RAM alone would hold.
- **Devices:** devices signal the processor with **interrupts** when they are ready or finished, so the processor never waits in a loop for them. Data can be buffered, and a device such as a printer can be shared by queuing jobs for it.

Hiding the hardware from the user

The user never has to know about memory addresses, disk sectors or the signals a printer needs. The OS puts layers in between:

- a **user interface**, usually a **graphical user interface (GUI)** with icons, windows and menus, so a user points and clicks instead of typing commands that refer to hardware;
- **device drivers**, which translate general requests ("print this page") into the exact signals a particular device needs, so every printer is used the same way;
- a **file system**, so data is seen as named files in folders, not as blocks on a disk;
- **device mapping**, so devices (local, network or virtual) appear under simple names, such as a drive letter, that can be checked and used easily.

Processes and multi-tasking

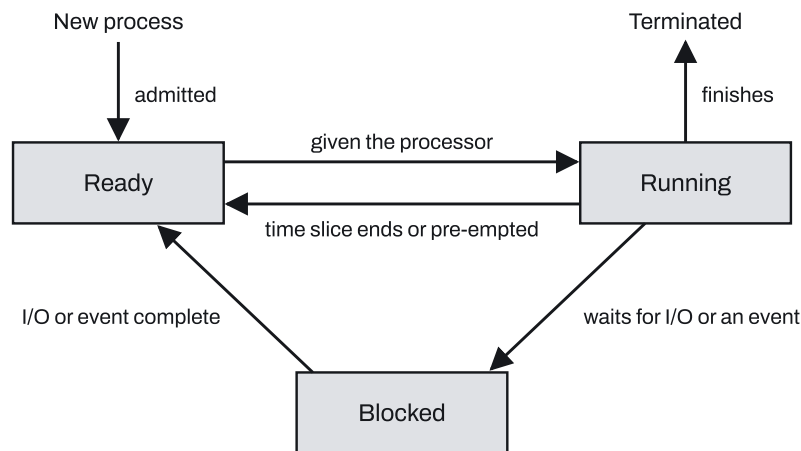
A **process** is a program that is being executed: its code plus its current data, register values and state. The OS keeps a record of each one, the **process control block (PCB)**, holding its ID, state, priority and saved registers.

Multi-tasking means running several processes so they **appear to run at the same time**. With one processor only one process actually runs at any instant, but the OS switches between them so quickly (many times a second) that they all seem to progress together. This lets more tasks finish than if each had to wait for the one before to end.

Process states

Every process is in one of three states:

- **Running:** it has the processor and is being executed. With one processor, only one process is running.
- **Ready:** it could run, and is waiting in the **ready queue** for its turn on the processor.
- **Blocked:** it cannot run, because it is waiting for something, such as an input/output operation to finish or an event to happen. Giving it the processor would be a waste.



The changes happen for these reasons:

Change	Why
ready → running	the low-level scheduler gives the process the processor
running → ready	its time slice ends (round robin), or a higher-priority process arrives and pre-empts it, or an interrupt takes the processor
running → blocked	it has to wait for input/output or another event
blocked → ready	the input/output or event it was waiting for is complete (the device signals this with an interrupt)

A blocked process never goes straight to running: it goes back to the ready queue first.

Scheduling

With many processes and one processor, something must decide which process runs next and for how long. That is **scheduling**. It is needed to:

- make multi-tasking possible, sharing the processor fairly between processes;
- keep the processor **busy all the time** (no idle time while a process waits for input/output);
- make sure every process is completed in a reasonable time and none is **starved** (never gets the processor);
- let high-priority jobs be done first, and get high throughput (many processes finished per second).

The **high-level scheduler** decides which programs are loaded into memory to become processes. The **low-level scheduler** (dispatcher) decides which ready process gets the processor next. A scheduling routine is **pre-emptive** if it can take the processor away from a running process, and **non-pre-emptive** if a process keeps the processor until it finishes or blocks.

The four scheduling routines:

Routine	How it works	Pre-emptive?	Benefit	Drawback
First come first served (FCFS)	processes are queued as they arrive and each runs to the end, in arrival order	no	simple, fair in order of arrival; no starvation	a long job holds up all the short jobs behind it, so the average waiting time can be long
Shortest job first (SJF)	from the queue, the process with the shortest total burst time (time it needs the processor) runs next, to the end	no	short jobs don't wait for long ones, so the average waiting time is lowest ; high throughput	long jobs can be starved if short ones keep arriving; burst times must be known
Shortest remaining time (SRT)	the process with the shortest time left runs; if a new process arrives needing less time than the running one has left, it replaces it	yes	short jobs finish very quickly; waiting time is minimised	long jobs can be starved; lots of switching
Round robin (RR)	processes are queued in arrival order, and each gets a fixed time slice (quantum) in turn; when the slice ends, the process's state is saved and it goes to the back of the queue, and the next process runs, carrying on from where it stopped last time	yes	every process gets a fair, equal share; good response time; no starvation	many context switches; long jobs take longer to finish

For example, four processes are all in the ready queue at time 0, in the order A, B, C, D, needing 5, 2, 7 and 3 ms of processor time.

Routine	Order the processor runs them	Average waiting time
FCFS	A (0–5), B (5–7), C (7–14), D (14–17)	6.5 ms
SJF	B (0–2), D (2–5), A (5–10), C (10–17)	4.25 ms
RR, 2 ms slices	A, B, C, D, A, C, D, A, C (B ends at 4, D at 13, A at 14, C at 17)	7.75 ms

(A process's waiting time is the time it finishes minus the time it needs, when all arrive at 0.) SJF gives the lowest average wait; round robin keeps every process moving.

SRT matters when processes arrive at different times. Process P arrives at time 0 needing 8 ms; Q arrives at time 2 needing 3 ms. With SJF, P keeps the processor to the end (0–8) and Q waits until 8. With SRT, at time 2 P has 6 ms left, more than Q's 3, so Q **pre-empts** P: P runs 0–2, Q runs 2–5, then P finishes 5–11.

The kernel and interrupts

The **kernel** is the central part of the OS that stays in memory and controls the processor, memory and devices. It is also the **interrupt handler**:

1. A device, the timer or a program sends an **interrupt** signal, and the kernel receives it.
2. The kernel checks the interrupt's **priority** against the process running now. If it is higher, the processor switches to kernel mode to deal with it; otherwise it waits.
3. The kernel **saves the state** of the interrupted process (the program counter and other registers) in its PCB or on the kernel stack.
4. It looks up the interrupt in the **interrupt dispatch table** (interrupt vector table) to find the address of the right **interrupt service routine (ISR)**, and runs it.
5. When the ISR finishes, the saved state is **restored**, and a process carries on.

This is how **low-level scheduling** works. In round robin, a **timer interrupt** ends each time slice: the kernel saves the running process, puts it back in the ready queue and dispatches the next one. When a disk finishes a read, its interrupt tells the kernel to move the waiting process from blocked to ready. Interrupts are handled **by priority**, so critical events are dealt with at once, and after an interrupt the scheduler may give the processor to a different process from the one that was interrupted.

Virtual memory

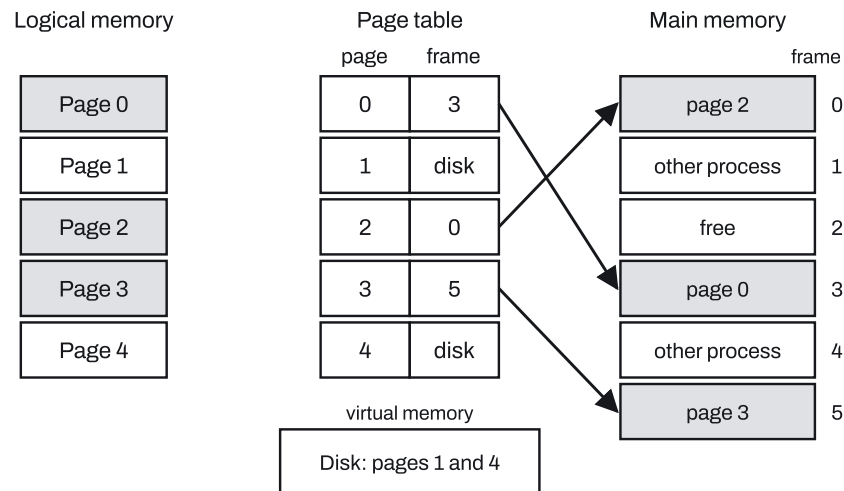
Programs are often bigger than the free RAM, and many run at once. **Virtual memory** uses part of **secondary storage** (the disk) as if it were extra main memory. Only the parts of a program that are needed right now are kept in RAM; the rest wait on the disk and are swapped in when needed. So:

- a program can be **larger than the RAM**;
- **more processes** can be in use at once than RAM alone would allow;
- RAM is used efficiently, holding only what is being used.

The cost is speed: the disk is much slower than RAM.

Paging

Paging is the usual way to do this. Main memory is split into fixed-size blocks called **page frames**, and each program (its logical memory) into blocks of **the same size** called **pages**. Any page can go into any free frame, so a program's pages need not sit together in RAM. A **page table** for each process records, for each page, whether it is in RAM and, if so, which frame holds it.



An address in the program is split into a **page number** and an **offset** (how far into the page). The page table turns the page number into a frame number; the offset stays the same.

If the process needs a page that is not in RAM, a **page fault** happens: the OS loads that page from the disk into a free frame. If there is no free frame, a page already in RAM must be sent back to the disk first. Choosing which one is **page replacement**:

- **First in, first out (FIFO):** replace the page that has been in RAM **longest**. Simple, but that page may still be in heavy use.
- **Least recently used (LRU):** replace the page that has gone **longest without being used**, since it is the least likely to be needed soon. Better choices, but the OS must record when each page was last used.

For example, with 3 frames and the pages used in the order 1, 2, 3, 1, 4, 1, 2, 5: FIFO causes 7 page faults and LRU 6. When 4 arrives, FIFO throws out page 1 (loaded first), which is needed again straight away; LRU throws out page 2, because page 1 was used more recently.

Segmentation

Segmentation also divides a program up, but into **segments of different sizes** that match the program's logical parts, such as a procedure, a module or an array. Each segment has a **number** (or name) and a **size**. A **segment map table** records, for each segment number, where the segment starts in physical memory and its size. An address is a **segment number and an offset**; the offset is checked against the segment's size and added to its start address. Segments are loaded from virtual memory into physical memory as they are needed.

	Paging	Segmentation
Size of the blocks	fixed , all the same	variable , each its own size
How they are chosen	the OS cuts memory up physically, ignoring the program's structure	follows the program's logical parts (the programmer or compiler decides)

	Paging	Segmentation
Table used	page table: page number → frame	segment map table: segment number → start address and size
Address	page number + offset	segment number + offset

Disk thrashing

With virtual memory, pages move between RAM and disk. If too many processes are running, or RAM is too full, a page is swapped out only to be needed again moments later, swapped in, and pushes out another page that is needed next. Then:

1. more and more pages are swapped in and out;
2. the disk is accessed constantly, with a very high rate of **read/write head movements**, and each takes a long time;
3. eventually the computer spends **more time swapping pages than processing** data: the **thrash point**. Programs slow right down or freeze.

It can be reduced by running fewer processes, adding RAM, or using a better page replacement routine.

How an interpreter runs a program

A compiler translates the whole program into object code first. An **interpreter** never produces a translated version. Instead, for **each statement in turn**:

1. it reads (fetches) the next statement of the **source code**;
2. it analyses it and checks it for errors;
3. if there is **no error**, it translates it and **executes it straight away** (by calling its own routines for that kind of statement);
4. if there **is an error**, it reports it and **stops** there.

Nothing translated is kept. So a statement inside a loop is translated **again every time round the loop**, and the whole program is translated again **every time it is run**. That is why interpreted programs run more slowly, and why the source code (and the interpreter) is needed each time.

The stages of compilation

A compiler turns the whole source program into object code in four main stages:

1. **Lexical analysis**. The first stage. It removes **white space and comments**, then splits the source code into **tokens**: keywords, identifiers, operators, numbers and so on. Each token is classified, often replaced by a short code, and identifiers go into a **symbol table** (with their type and, later, their address). A character sequence that isn't a valid token is reported as an error. The stream of tokens goes to the syntax analyser.

2. **Syntax analysis** (parsing). The tokens are checked against the **grammar rules** of the language (the rules written in BNF or syntax diagrams). Any **syntax errors** are reported. If the code is correct, the output is a **parse tree** (or an intermediate code) showing the structure of each statement.
3. **Code generation**. The parse tree or intermediate code is turned into **object code** (machine code) that the processor can run.
4. **Optimisation**. The code is improved so that it uses **fewer resources**: it runs in less time, uses less memory and takes less storage. For example, it removes code that is never used or does nothing, works out constant expressions such as $3 * 4$ once instead of at every run, and reorders instructions. (Optimisation can happen during code generation too.)

Syntax diagrams and BNF

Every language has **grammar rules** that say which sequences of characters are allowed. The syntax analyser checks code against them. Two ways to write them:

Backus-Naur Form (BNF). Each rule has the form `<name> ::= definition`:

- `<letter>` in angle brackets is a **non-terminal**: a name for something defined by another rule;
- a character written as itself, such as A or =, is a **terminal**: it appears in the code exactly like that;
- `::=` means "is defined as" and `|` means "or";
- elements written next to each other come one after the other.

```
<letter> ::= A | B | C
<digit>  ::= 0 | 1 | 2 | 3
<code>   ::= <letter><digit> | <letter><digit><digit>
```

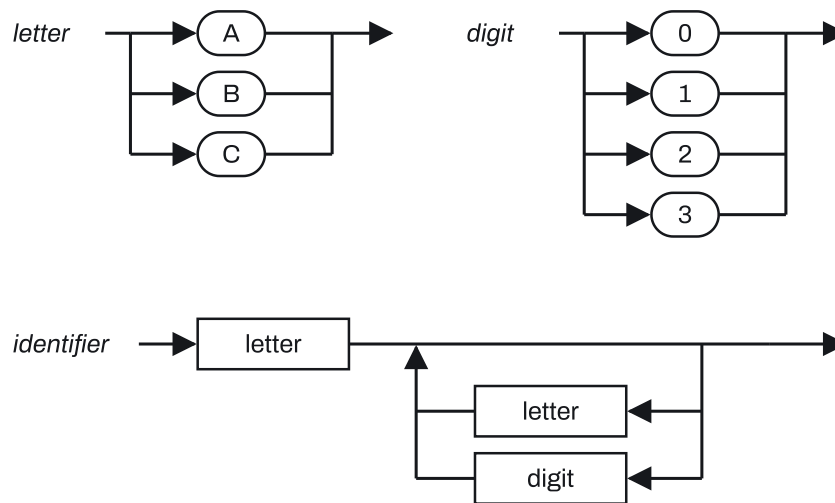
A `<code>` is a letter followed by one or two digits, such as B3 or C20.

BNF has no symbol for "repeat". To allow **one or more** of something, a rule calls **itself** (it is **recursive**):

```
<digits> ::= <digit> | <digit><digits>
```

A `<digits>` is one digit, or one digit followed by more digits. The single `<digit>` option is essential: it lets the recursion stop.

Syntax diagrams. The same rules as pictures. You follow the arrows from the rule's name on the left to the exit on the right. Each box you pass is one element; a **rectangle** holds a non-terminal and a **rounded box** holds a terminal. Where the line **splits**, you choose one branch (that is `|`). A line that **loops back** lets you repeat, and a line that **bypasses** a box makes it optional.



The identifier diagram means: one letter, then **any number** (including none) of letters or digits. So A, BC31 and C0A2 are valid; 2B is not, because it starts with a digit. In BNF:

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

To check a string against a diagram, walk through it one character at a time: at each point the character must match one of the boxes you could reach next, and when the string ends you must be able to reach the exit.

Reverse Polish Notation

We normally write expressions in **infix** form: the operator goes **between** its operands, as in $(a + b) \times c$. That needs brackets and precedence rules (multiplication before addition) to be unambiguous. In **Reverse Polish Notation (RPN)**, or **postfix**, each operator goes **after** its two operands:

Infix	RPN
$a + b$	a b +
$(a + b) \times c$	a b + c *
$a + b \times c$	a b c * +
$(a - b) \times (c + d)$	a b - c d + *

RPN needs **no brackets and no precedence rules**: the order of the operators says exactly what is done when. It is **unambiguous** and can be worked out in **one pass from left to right** using a **stack**. That makes it easy for a compiler or interpreter to turn expressions into simple instructions, and it is how many compilers handle expressions.

Evaluating RPN with a stack. Read the expression from left to right:

1. an **operand** (a number or a variable's value): **push** it onto the stack;
2. an **operator**: **pop** the top two values, apply the operator (the **second** value popped is on the **left**: first popped y , second popped x gives $x - y$ or $x \div y$), and **push** the result;

3. at the end, the one value left on the stack is the answer.

Operators are never put on the stack. For $5 \ 1 \ + \ 8 \ 2 \ / \ *$, the stack after each change is (the top is the highest filled row):

	5	1	+	8	2	/	*
third					2		
second		1		8	8	4	
bottom	5	5	6	6	6	6	24

So the value is 24, which is $(5 + 1) \times (8 \div 2)$.

A **binary tree** can also hold an expression: operators at the nodes, operands at the leaves. Its post-order traversal gives the RPN. The stack is the usual answer for **evaluating** it. Stacks are also used for procedure calls (storing return addresses), recursion and saving registers during interrupts.

Key facts and formulas

There is no formula list in the 9618 exams, so every fact here is **Learn it**.

Formula	Status
Process states: running, ready, blocked. Ready $\rightarrow$ running (given the processor); running $\rightarrow$ ready (time slice ends or pre-empted); running $\rightarrow$ blocked (waits for I/O or event); blocked $\rightarrow$ ready (event complete)	Learn it
FCFS: arrival order, to the end. SJF: shortest burst first, to the end (non-pre-emptive). SRT: shortest time left, pre-emptive. Round robin: equal time slices in turn, pre-emptive	Learn it
Kernel as interrupt handler: receive, check priority, save state, find ISR in the interrupt dispatch table, run it, restore state	Learn it
Paging: fixed-size pages and frames, page table, page number + offset. Segmentation: variable-size logical segments, segment map table, segment number + offset	Learn it
Page replacement: FIFO (oldest page) or LRU (least recently used page)	Learn it
Disk thrashing: pages swapped in and out so often that more time is spent swapping than processing	Learn it
Compilation: lexical analysis $\rightarrow$ syntax analysis $\rightarrow$ code generation $\rightarrow$ optimisation	Learn it
BNF: $::=$ is defined as, or, <name> non-terminal; repetition by recursion, such as <digits> $::=$ <digit> <digit><digits>	Learn it

Formula	Status
RPN with a stack: push operands; for an operator pop two, apply (second popped on the left), push the result	Learn it

How to do it

In the Paper 3 questions we have tagged for this topic (52 questions, 2021–2025, counting a question each time it was set, as many were set on two or three papers), the types came up this often. 11 of the 52 questions mix types, so the counts add up to more than 52.

Question type	Questions
Reverse Polish Notation: convert and evaluate	16
Syntax diagrams and BNF	12
Multi-tasking, process states and interrupt handling	10
Stages of compilation	8
Virtual memory, paging, segmentation and disk thrashing	8
Scheduling routines	5
How an interpreter executes a program	4
Hiding the hardware from the user	3

1. Reverse Polish Notation: convert and evaluate (16 questions)

Infix to RPN:

- Put brackets round every operation in the order it is done: brackets first, then $\times$ and $\div$, then $+$ and $-$, and **left to right** for operators of equal rank. So $a \times b + b - d + 15$ is $((a \times b) + b) - d + 15$.
- Starting with the innermost bracket, write its two operands then its operator. Keep the operands in their original **left-to-right order**; only the operators move.
- Check: the number of operators is one less than the number of operands, and there are no brackets.

$a \times b + b - d + 15$ becomes $a \ b \ * \ b \ + \ d \ - \ 15 \ +$. And $(7 - 2) \times (4 + 6) \div 5$ becomes $7 \ 2 \ - \ 4 \ 6 \ + \ * \ 5 \ /$.

RPN to infix: work left to right with a stack of expressions. Push operands; at an operator, pop two, write "(left operator right)" with the **second-popped on the left**, and push that. At the end remove brackets that aren't needed. $a \ b \ - \ c \ d \ + \ * \ a \ /$ gives $((a - b) \times (c + d)) \div a$.

Evaluate and show the stack (the paper gives an empty stack drawn as columns):

- Replace each variable by its value (write the values above the letters so you don't mix them up).

2. Fill one column each time the stack **changes**: after each push, and after each operator (the two values are gone and the result is there). Values stay at the bottom and build upwards; a column shows the **whole** stack, not just the new value.
3. Never write an operator in the stack.
4. The last column holds one value: the answer. Check it by working out the infix form.

Variations you will meet:

- **"Explain how a stack is used to evaluate RPN"**: read left to right; push each operand; at an operator pop the top two, apply the operator, push the result; carry on to the end; the answer is the single value left. Better answers use the actual expression: "push 10, push 3, push 2; at * pop 2 and 3, push 6 ...".
- **"Why is RPN used?"**: no brackets or precedence rules, so it is unambiguous; it is evaluated left to right in one pass with a stack; easy for a compiler to turn into machine code. Answer about RPN here, not about stacks.
- **"Identify a data structure, with reasons"**: a stack, because the last operands pushed are the first needed by the next operator (LIFO); or a binary tree, traversed in post-order.
- **"State two other uses of a stack"**: procedure calls and return addresses, recursion, saving registers during interrupt handling.
- **Evaluate an infix answer**: substitute the values, following precedence.

2. Syntax diagrams and BNF (12 questions)

"State why ... is invalid": walk through the diagram with the string (see "Syntax diagrams and BNF") and name the **first** rule it breaks, using the diagram's words: "a passcode must begin with an upper, not a symbol", "the second character must be a lower or a digit", "there can be only one or two digits, not three", "? is not one of the symbols". Say **what** is wrong and **what** was allowed.

Diagram to BNF:

1. Single-choice rules (letter, digit, symbol, operator): list every terminal on the diagram, separated by | :
`<operator> ::= + | - | * | /.`
2. A fixed sequence with choices at some points: either list **every combination** in full, joined by | , or make a helper rule for the choice and use it (`<middle> ::= <lower> | <digit>`), then `<passcode> ::= <upper><middle><middle><digit>`).
3. A loop ("one or more", "can be repeated") needs a **recursive** rule with a stopping option: `<code> ::= <digit> | <digit><code>`. "Zero or more" means also listing the version without it.
4. Copy the names exactly as the paper writes them, with `< >`, `::=` and `|`, and use terminals only from the diagrams given.

Drawing a syntax diagram: write the rule's **name** at the start; draw the elements left to right in order, each in a box; make a **split** where there is a choice (a parallel branch for each option), a **line going back** before a box for repetition, and a **line going round** a box to make it optional; add arrows so there is one way in and one way out, and no extra boxes.

- "One or two numbers": a number box, then a second number box with a bypass round it.
- "One or more letters": a letter box with a loop back round it.
- "The last character may be repeated": a loop back round the last box.

"Give an example of a valid ...": build it from the terminals actually on the diagrams and check every constraint, such as "at least four characters".

3. Multi-tasking, process states and interrupt handling (10 questions)

- **Multi-tasking**: several processes appear to run at the same time; processor time and other resources are **shared** between them; **scheduling** decides which runs, so they use the hardware efficiently and don't clash; a higher-priority task can interrupt the one running. The OS ensures it runs efficiently by monitoring the state of each process and scheduling them.
- **Name the states**: running, ready, blocked. **Reason for each**: running, it has been given processor time and is being executed; ready, it is waiting in the ready queue for a time slice (it could run); blocked, it is waiting for input/output or an event to finish.
- **Running** → **ready**: its time slice ends; or a higher-priority process in the ready queue pre-empts it; or an interrupt arrives.
- **Kernel as interrupt handler / interrupt handling in low-level scheduling**: give the steps from "The kernel and interrupts": the kernel receives the signal, checks priority, saves the state of the current process, uses the interrupt dispatch table to find the ISR, runs it, then restores the state; interrupts are dealt with by priority so critical events are handled at once.
- **Match the term to its description**: multi-tasking (many programs appear to run at once), scheduling (managing the processes on the processor), interrupt handling (passing control to another routine when a service is needed), virtual memory (secondary storage used as extra main memory), paging (reading and writing same-size blocks between memory and secondary storage).

4. Stages of compilation (8 questions)

1. **Name the stages** in order: lexical analysis, syntax analysis, code generation, optimisation. "Two other stages" than lexical and syntax analysis means code generation and optimisation.
2. **Describe a stage**: give 3–4 separate facts from "The stages of compilation" for that stage: its place in the order, what goes in, what it does, what comes out, and what errors it reports.
 - **Lexical**: first stage; white space and comments removed; source split into tokens, each classified; identifiers put in the symbol table; invalid tokens reported; tokens passed to the syntax analyser.
 - **Syntax**: second stage; takes the token stream; checks it against the grammar rules of the language; reports syntax errors; outputs a parse tree.
 - **Optimisation**: removes redundant code, simplifies expressions or reorders code, **so that** execution time, memory and storage are minimised. A 1-mark "purpose" answer: to make the code use as few resources (processor time, memory, storage) as possible.

3. **Matching:** lexical analysis ↔ characters into tokens; syntax analysis ↔ parsing the tokens to check their structure; code generation ↔ intermediate code into an executable form; optimisation ↔ reducing execution time and memory needs. (Running a script statement by statement is interpretation, not a stage of compilation.)

5. Virtual memory, paging, segmentation and disk thrashing (8 questions)

- **Virtual memory** (what and why): part of secondary storage used as if it were main memory; used when RAM runs low, such as when many processes are running; only the pages needed are kept in RAM, so programs bigger than RAM can run and RAM is used efficiently.
- **Describe paging:** fixed-size pages and frames, page table, page number + offset, pages loaded when needed (page fault), page replacement.
- **Describe segmentation:** logical memory split into **variable-sized** segments; each has a number (name) and size; segments loaded from virtual memory into physical memory as needed; the address is a segment number plus offset; a **segment map table** maps each segment number to its physical address and size.
- **One difference:** state **both** sides, for example "paging divides memory into **fixed-size** blocks, while segmentation divides it into **variable-size** blocks".
- **Disk thrashing:** happens when virtual memory is used heavily; as main memory fills, more and more pages are swapped in and out; the disk head moves constantly, and each move is slow; eventually more time is spent swapping pages than processing, so programs freeze.

6. Scheduling routines (5 questions)

1. **Why scheduling is needed:** give separate points: allows multi-tasking; keeps the processor busy (minimum idle time); every process is completed in reasonable time with fair access, and none is starved; higher-priority jobs are done first.
2. **Name the routines:** round robin, shortest job first, first come first served, shortest remaining time.
3. **Describe a routine and give a benefit:** say more than its name. Cover: how processes join the queue (as they arrive), which one is chosen next, whether it is **pre-emptive**, and what happens at the end of a slice or when a shorter job arrives; then one benefit. Use the table in "Scheduling".
 - **Round robin:** queued as they arrive; each gets an equal, fixed time slice in turn; pre-emptive; at the end of a slice the state is saved and the process rejoins the queue, carrying on where it left off next time; a process that finishes early lets the next one run. Benefit: no starvation, good average response time.
 - **SJF:** queued as they arrive; the one with the shortest burst time runs next; non-pre-emptive, it runs until it finishes or blocks; long jobs can be starved. Benefit: short jobs don't wait for long ones, so the average waiting time is lowest; high throughput.
 - **SRT:** queued as they arrive; shortest burst (remaining) time runs first; pre-emptive, a newly arrived shorter process replaces the running one; long jobs can be starved. Benefit: short processes finish very quickly, waiting time is minimised.

- **FCFS**: queued in arrival order; each runs to completion; non-pre-emptive. Benefit: simple, no starvation.

7. How an interpreter executes a program (4 questions)

Give the steps from "How an interpreter runs a program": one statement at a time; check it for errors; if error-free, translate and **execute it at once**; if there's an error, report it and halt; no executable/object code is stored; each statement is translated again every time it is run, including every pass of a loop. Describe what the interpreter **does**, not its benefits or a compiler's steps.

8. Hiding the hardware from the user (3 questions)

Name a way **and** describe it: a (graphical) user interface, with icons and menus so the user points and clicks rather than using commands about memory or hardware; device drivers, so peripherals such as printers are controlled through the OS in the same way; device mapping, so devices can be found and used by simple names; the user works only with the application layer, and the lower layers stay hidden. A "benefit and example" question: the benefit is that the user doesn't need to know how the hardware works (with an example such as clicking an icon to open a file instead of typing a command).

Worked examples

The bold codes show where marks are earned, as in Cambridge mark schemes. The number is how many marks.

- **B1** mark for a correct result on its own, with no method needed.
-

Example 1

(a) Write the infix expression $(7 - 2) \times (4 + 6) \div 5$ in Reverse Polish Notation. **[2]**

(b) The RPN expression $a \ b \ c \ + \ * \ d \ /$ is to be evaluated, where $a = 6$, $b = 5$, $c = 7$ and $d = 4$. Show the changing contents of the stack as it is evaluated. **[4]**

(c) Write the RPN expression $x \ y \ - \ z \ x \ + \ *$ in infix form. **[2]**

(a) Bracketed in order: $(((7 - 2) \times (4 + 6)) \div 5)$. **B1** for $7 \ 2 \ - \ 4 \ 6 \ +$, **B1** for the rest:

$7 \ 2 \ - \ 4 \ 6 \ + \ * \ 5 \ /$

(b) With the values: $6 \ 5 \ 7 \ + \ * \ 4 \ /$.

	6	5	7	+	*	4	/
third			7				

	6	5	7	+	*	4	/
second		5	5	12		4	
bottom	6	6	6	6	72	72	18

At + the 7 and 5 are popped and $5 + 7 = 12$ pushed; at *, $6 \times 12 = 72$; at /, $72 \div 4 = 18$. **B1** for each pair of correct columns (up to **B4**), with the final value 18.

(c) Push x , y ; at $-$ pop and push $(x - y)$; push z , x ; at $+$ push $(z + x)$; at $*$ pop both:

$$(x - y) \times (z + x)$$

B1 for each bracket, both joined by $\times$.

Example 2

A code in a game is defined by these syntax rules:

```
<letter> ::= P | Q | R
<digit>  ::= 1 | 2 | 3 | 4
<symbol> ::= # | &
```

A **tag** is one letter, followed by one or two digits, followed by one symbol.

(a) State whether each of $R23\&$, $PQ1\#$ and $Q4\&\#$ is a valid tag, with a reason. **[3]**

(b) Write the BNF for $\langle \text{tag} \rangle$. **[2]**

(c) A **key** is one or more letters followed by a digit. Draw a syntax diagram for key and write its BNF. **[4]**

(a)

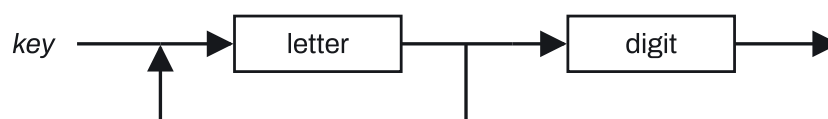
- $R23\&$: **valid**: a letter, two digits, then a symbol. **B1**
- $PQ1\#$: **invalid**: the second character must be a digit, but Q is a letter (only one letter is allowed). **B1**
- $Q4\&\#$: **invalid**: a tag ends with exactly one symbol, and this has two. **B1**

(b) One option for one digit, one for two, joined by "or":

```
<tag> ::= <letter><digit><symbol> | <letter><digit><digit><symbol>
```

B1 for each option.

(c) "One or more letters" needs a loop round the letter box:



B1 for the letter box with a loop that allows repeats, **B1** for one digit at the end with the name and arrows correct.

In BNF the repeat is a recursive helper rule:

```
<letters> ::= <letter> | <letter><letters>
<key> ::= <letters><digit>
```

B1 for <letters> with both the single option and the recursive one, **B1** for <key>. So QRP3 and P1 are keys, but 1P is not.

Example 3

A computer uses multi-tasking. Four processes are in the ready queue at time 0, in the order A, B, C, D. They need 5, 2, 7 and 3 ms of processor time.

- (a) Identify the three process states and give a reason why a process could be in the blocked state. **[2]**
- (b) Give the order in which the processes run with round robin and 2 ms time slices, and the time at which each finishes. **[3]**
- (c) Describe the shortest job first scheduling routine and give one benefit of it. Use the processes above to support your answer. **[4]**

(a) Running, ready and blocked. **B1** A process is blocked when it is waiting for an input/output operation or another event to be completed. **B1**

(b) A (0–2), B (2–4), C (4–6), D (6–8), A (8–10), C (10–12), D (12–13), A (13–14), C (14–17). **B1** for the order of the first round, **B1** for the repeats with the shorter final slices. B finishes at 4, D at 13, A at 14 and C at 17. **B1**

(c) Any three of: processes are queued as they arrive (**B1**); the process with the shortest burst time is chosen next and executed (**B1**), so here the order is B, D, A, C; it is non-pre-emptive: once started, a process runs until it finishes or has to wait (**B1**); long jobs can be starved if shorter ones keep arriving (**B1**). Benefit: short jobs don't wait for long ones, so the average waiting time is the lowest possible: here 4.25 ms, compared with 7.75 ms for round robin and 6.5 ms for first come first served (**B1**).

Example 4

- (a) Explain what is meant by virtual memory, and why an operating system uses it. **[3]**
- (b) Describe how paging is used to manage memory, and state one difference between paging and segmentation. **[4]**
- (c) Explain how disk thrashing can occur. **[3]**

(a) Any three of: part of secondary storage is used as if it were extra main memory (**B1**); it is used when RAM is running low, for example with many processes running (**B1**); only the parts of programs currently needed are kept in RAM, the rest on disk (**B1**); so programs larger than RAM can run and RAM is used efficiently (**B1**).

(b) Main memory is divided into fixed-size page frames and each program into pages of the same size (**B1**); a page table records which frame holds each page (**B1**); when a page not in RAM is needed it is loaded from disk, replacing a page chosen by a routine such as least recently used (**B1**). Difference: pages are all the same fixed size, but segments have variable sizes that match the logical parts of the program (**B1**).

(c) Any three of: thrashing happens when virtual memory is in use (**B1**); as main memory fills, more and more pages are swapped in and out between RAM and disk (**B1**); this means a very high rate of disk access, and the read/write head movements are slow (**B1**); eventually more time is spent swapping pages than processing, so the program freezes or runs very slowly (**B1**).

Common mistakes

- **Operators on the stack.** Examiners don't allow operators in the stack columns. Only numbers go on the stack; an operator pops two and pushes the result.
- **Swapping the operands** when you pop. For $12 \div 3$ / the first value popped is 3 and the second 12, so the result is $12 \div 3 = 4$, not $3 \div 12$. The same applies to $-$.
- **Changing the order of operands** when converting to RPN. Only the operators move; $a - b$ is $a \ b \ -$, never $b \ a \ -$. For equal-rank operators work left to right: $a - b + c$ is $a \ b \ - \ c \ +$.
- **Answering the wrong question on RPN.** "Why is RPN used" is about RPN (no brackets, unambiguous, left to right), not about how a stack works; that is a separate question.
- **BNF errors.** Reports mention answers that show the right content but break the notation: missing $< > , =$ instead of $: :=$, commas instead of $|$. For "one or more" you need a **second, recursive rule** called by the first; few candidates saw this.
- **Syntax diagrams that allow too much or too little:** a loop where only "one or two" is allowed, no bypass where something is optional, or boxes in the wrong order. Check your diagram with a valid and an invalid example.
- **Just repeating the name of a scheduling routine.** "Shortest job first runs the shortest job first" earns nothing. Say how processes are queued, how the next one is chosen, whether it is pre-emptive, and give a real benefit.
- **Paging and segmentation the wrong way round,** or describing only one of them when asked for a **difference**. Paging is fixed-size, segmentation variable-size; state both halves.
- **Virtual memory vs virtual machines.** Some candidates wrote about virtual machines when asked about virtual memory.
- **Describing a compiler, or the benefits of an interpreter,** when asked how an interpreter executes a program. The key point is that each statement is translated and **executed immediately**, and nothing translated is kept.
- **Vague compilation answers.** For syntax analysis, "checks the code" is not enough: say it checks the tokens against the **grammar rules** of the language, reports syntax errors and produces a parse tree.

How to get the marks

- **Stack columns are marked in rings** (small groups of consecutive columns), usually one mark per group plus one for the key values such as the final answer. Show **every** change of the stack, with the full contents each time; one wrong column costs only that group, so keep going.
- **RPN conversions** get a mark for each correct part and one for the complete answer. Write it with spaces between the tokens.
- **"Show the changing contents of the stack"** means the columns, not just the answer. **"Show your working"** can be the stack or the infix form with values substituted.
- **BNF** is marked by rule and by option: each correct option in a list of alternatives, and the recursive rule, can earn marks. Use the paper's names and terminals exactly.
- **Syntax diagrams**: marks for the right boxes in the right order, for the branch or loop that gives the choice or repeat, and for the name, arrows and no extra boxes.
- **"Identify" or "State"**: short answers (one state, one routine name). **"Describe"**: what it is and how it works (for scheduling: queue, choice, pre-emption). **"Explain"**: reasons, with "so" or "because" (why scheduling is needed, why thrashing happens). **"Outline"**: brief points in order (the interrupt-handling steps).
- **"Describe ... and give a benefit"**: the benefit is marked separately and must be a real advantage (lower average waiting time, no starvation), not a restatement of how it works.
- **Mark points build on each other** ("... and so ..."): for thrashing, give the chain in order: virtual memory in use → pages swapped in and out more and more → constant slow head movement → more time swapping than processing.

Quick check

1. Write $(a + b) \times c - d \div e$ in RPN.
2. Evaluate the RPN expression $9 \ 3 \ - \ 4 \ 2 \ + \ * \ 12 \ /$, showing the stack after each change.
3. A `<number>` is one or more digits. Write its BNF, given `<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9`.
4. A process has 3 page frames and uses pages in the order 1, 2, 3, 1, 4, 1, 2, 5. All frames start empty. Count the page faults with FIFO replacement and with LRU replacement.
5. Name the stage of compilation that: (a) builds the symbol table; (b) produces a parse tree; (c) removes code that is never used.

Answers

1. Bracketed in order: $((a + b) \times c) - (d \div e)$, so a b + c * d e / -.
2. Push 9, push 3; - gives $9 - 3 = 6$; push 4, push 2; + gives 6; * gives $6 \times 6 = 36$; push 12; / gives $36 \div 12 = 3$. The stack (bottom first) goes: $9 \rightarrow 9, 3 \rightarrow 6 \rightarrow 6, 4 \rightarrow 6, 4, 2 \rightarrow 6, 6 \rightarrow 36 \rightarrow 36, 12 \rightarrow 3$. The value is 3.
3. $\langle \text{number} \rangle ::= \langle \text{digit} \rangle \mid \langle \text{digit} \rangle \langle \text{number} \rangle$ (or $\langle \text{number} \rangle \langle \text{digit} \rangle$ as the recursive option).
4. **FIFO: 7 faults.** 1, 2, 3 fault (frames 1, 2, 3); 1 is there; 4 faults and replaces 1 (the oldest); 1 faults and replaces 2; 2 faults and replaces 3; 5 faults and replaces 4. **LRU: 6 faults.** 1, 2, 3 fault; 1 is there; 4 faults and replaces 2 (least recently used); 1 is there; 2 faults and replaces 3; 5 faults and replaces 4.
5. (a) Lexical analysis. (b) Syntax analysis. (c) Optimisation.

Past questions to try

- [9618/32/M/J/24 Q5](#): infix to RPN, evaluating with a stack, then RPN back to infix.
- [9618/32/M/J/23 Q3](#): checking strings against syntax diagrams, recursive BNF and drawing a diagram.
- [9618/32/O/N/22 Q9](#): why scheduling is needed, and three scheduling routines with their benefits.
- [9618/32/O/N/23 Q5](#): why virtual memory is used, and when disk thrashing occurs.