

Topic 16: System Software

How a compiler's front end really works, and how languages reclaim memory automatically.

Optional stretch material. This goes beyond the 9618 syllabus, so you will not be asked about it directly in the exam. Its purpose is to deepen ideas you already know. **Builds on:** 16.2 Language translators (lexical analysis, syntax analysis, BNF). Try the challenge questions on the last pages before you read the answers.

Lexical analysis with finite automata

The first compiler stage turns characters into **tokens**. Each token type can be described by a *pattern* and recognised by a small finite state machine (as in the topic 15 extension):

Token type	Pattern	Examples
Identifier	letter followed by letters or digits	total, x1, Count
Integer	one or more digits	7, 120
Real	digits, a point, digits	3.14
Operators	single symbols	+ - * = ()

The lexer reads the longest string that matches a pattern (so count1 is one identifier, not count and 1), skips white space and comments, and passes the token stream to the parser. This is the same idea as a *regular expression*.

Recursive descent parsing

A parser checks that tokens follow the grammar (BNF). A beautifully simple method, **recursive descent**, turns each BNF rule into one function. Here is a grammar for arithmetic, with brackets and correct operator precedence:

```
<expr> ::= <term> { ("+" | "-") <term> }
<term> ::= <factor> { ("*" | "/" ) <factor> }
<factor> ::= <number> | "(" <expr> ")"
```

(Braces mean "zero or more times".) Each rule becomes a function, for example:

```
FUNCTION ParseExpr()
  ParseTerm()
  WHILE NextToken = "+" OR NextToken = "-"
    ReadToken()
    ParseTerm()
  ENDWHILE
ENDFUNCTION
```

ParseFactor calls ParseExpr for bracketed input, so the functions call each other recursively exactly as the grammar does. Precedence comes free: * is handled deeper (in term) than +, so in $2 + 3 \times 4$ the product is grouped first. A rule like $\langle \text{expr} \rangle ::= \langle \text{expr} \rangle + \langle \text{term} \rangle$ (left recursion) would make this method loop forever, so grammars are rewritten first.

Garbage collection

Languages such as Java and Python free memory automatically. Two classic approaches:

Method	How it works	Weakness
Reference counting	Each object counts references to it; at zero it is freed immediately.	Cycles: A refers to B and B to A keep both counts at 1 forever, so they leak.
Mark and sweep	Start from roots (variables on the stack); mark everything reachable; sweep (free) the rest.	Program may pause while collecting; cost grows with the amount of live data.

Mark-and-sweep handles cycles, because unreachable objects are never marked however they refer to each other.

Challenge questions

1. List the tokens (with their types) in `total = 3.5 * (x + 12)`.
2. Explain how the grammar `<expr>/<term>/<factor>` above makes multiplication bind more tightly than addition.
3. Objects A and B refer to each other and nothing else refers to either. Why does reference counting fail to free them, and how does mark and sweep succeed?

Worked answers

Only look once you have had a proper go at each question.

1. identifier total; operator =; real 3.5; operator *; operator (; identifier x; operator +; integer 12; operator).
2. An <expr> is a sum of <term>s, and each <term> is a product of <factor>s. In $2 + 3 * 4$ the parser sees the terms "2" and " $3 * 4$ "; the multiplication is completed inside the second <term> before the addition combines the terms, so the result is $2 + (3 \times 4) = 14$.
3. Each object has a reference count of 1 (from the other), so neither ever reaches zero. Mark and sweep marks only objects reachable from the roots; A and B are not reachable, so they stay unmarked and are freed in the sweep.