Compiler Design

Short notes for B.Tech CSE / IT · MCA · GATE CS

Uncodemy Handwritten Notes · free for students

I. Introduction

- **Compiler** = translates a whole high-level program into target (machine / assembly) code.
- **Interpreter** = translates and runs one statement at a time.
- **Assembler**: assembly → machine code. **Linker**: joins object files. **Loader**: loads into memory.

➤ Phases of a compiler

```
source code
  -> Lexical analysis       (tokens)
  -> Syntax analysis        (parse tree)
  -> Semantic analysis      (type checks, annotated tree)
  -> Intermediate code gen  (three-address code)
  -> Code optimization
  -> Target code generation
Symbol table and error handler work with all phases
```

- **Front end** (analysis): lexical, syntax, semantic, intermediate code. Depends on the source language.
- **Back end** (synthesis): optimization, code generation. Depends on the target machine.
- **Pass** = one reading of the program. One-pass compilers are fast; multi-pass give better optimization.

Compiler	Interpreter
Whole program at once	Line by line
Faster execution	Slower execution
Errors shown after compiling	Stops at first error
C, C++	Python (CPython uses bytecode + interpreter)

2. Lexical Analysis

- Reads characters and groups them into **lexemes**, then gives **tokens** (keyword, identifier, operator, constant, punctuation).
- Removes whitespace and comments. Enters identifiers into the symbol table.
- Tokens are described with **regular expressions** and recognised with a `DFA`. Tool: LEX / Flex.
- Longest match rule: `<=` is read as one token, not `<` then `=`.

EXAMPLE Count tokens in `int a = b + 10;` → int, a, =, b, +, 10, ; = `7 tokens`.

TIP In GATE token-count questions, a string literal like "hello %d" is one token, and comments are not tokens.

3. Syntax Analysis (Parsing)

- Checks if the token stream follows the grammar (CFG). Builds a parse tree.

➤ Parser types

Top-down	Bottom-up
Starts from the start symbol	Starts from the input string
Leftmost derivation	Reverse of `rightmost derivation`
Recursive descent, LL(1)	Shift-reduce: LR(0), SLR, LALR, CLR
Cannot handle left recursion	Handles left recursion

➤ Removing left recursion and left factoring

- $A \to A\alpha \mid \beta$ becomes $A \to \beta A'$, $A' \to \alpha A' \mid \varepsilon$.
- $A \to \alpha\beta 1 \mid \alpha\beta 2$ becomes $A \to \alpha A'$, $A' \to \beta 1 \mid \beta 2$ (left factoring).

EXAMPLE $E \to E + T \mid T$ becomes $E \to T\ E'$, $E' \to +\ T\ E' \mid \varepsilon$.

➤ FIRST and FOLLOW

- **FIRST(X)** = terminals that can start strings derived from X (include ε if X can give ε).
- **FOLLOW(A)** = terminals that can come right after A. FOLLOW(start) contains $.

Solved example

- Grammar: E → T E', E' → + T E' | ε, T → F T', T' → * F T' | ε, F → (E) | id.

Symbol	FIRST	FOLLOW
E	{ (, id }	{), $ }
E'	{ +, ε }	{), $ }
T	{ (, id }	{ +,), $ }
T'	{ *, ε }	{ +,), $ }
F	{ (, id }	{ *, +,), $ }

LL(1)

- Table M[A, a]: for A → α, put it in M[A, a] for each a in FIRST(α). If ε is in FIRST(α), put it under each b in FOLLOW(A).
- Grammar is LL(1) if no table cell has more than one entry.
- Left-recursive or ambiguous grammars are never LL(1).

4. Bottom-Up Parsing

- **Shift-reduce parser**: uses a stack. Actions: shift, reduce, accept, error.
- **Handle** = the substring that matches a production's right side and is reduced next.
- Conflicts: **shift-reduce** and **reduce-reduce**.

LR parsers

Parser	Items used	Reduce placed under	Power
LR(0)	LR(0) items	All terminals	Weakest
SLR(1)	LR(0) items	FOLLOW of left side	More than LR(0)
LALR(1)	Merged LR(1) items	Lookahead symbols	Used by YACC
CLR(1) / LR(1)	LR(1) items	Lookahead symbols	Most powerful

REMEMBER Power: LR(0) < SLR(1) < LALR(1) < CLR(1). Number of states: LR(0) = SLR = LALR <= CLR.

- LALR is made by merging CLR states with the same core. Merging can create reduce-reduce conflicts, but never new shift-reduce conflicts.
- Every LL(1) grammar is LR(1). Every SLR grammar is LALR.

- **Operator precedence parser**: for operator grammars (no ε, no two adjacent variables).

➤ LR(0) item example

- Production A → XY gives items: A → .XY, A → X.Y, A → XY.
- Closure: if A → α.Bβ is an item, add B → .γ for each production of B.

5. Syntax-Directed Translation (SDT)

- Attach **semantic rules** (actions) to grammar productions.
- **Synthesized attribute**: value comes from children. **Inherited attribute**: from parent or siblings.
- **S-attributed SDD**: only synthesized attributes. Works with bottom-up (LR) parsing.
- **L-attributed SDD**: inherited attributes only from parent or left siblings. Works with top-down parsing. Every S-attributed is L-attributed.

> **EXAMPLE** E → E1 + T { E.val = E1.val + T.val }. For 3 + 4 * 5 with normal precedence, value = 23.

6. Intermediate Code Generation

- Machine-independent code between source and target. Makes the compiler easy to retarget.
- Forms: syntax tree, DAG, postfix, three-address code (TAC).

➤ Three-address code for a = b * -c + b * -c

```
t1 = -c
t2 = b * t1
t3 = -c
t4 = b * t3
t5 = t2 + t4
a  = t5
```

- With a **DAG**, common sub-expressions are shared: t1 = -c, t2 = b * t1, a = t2 + t2.

➤ Representations of TAC

Form	Fields	Note
Quadruple	op, arg1, arg2, result	Easy to move code
Triple	op, arg1, arg2	Result referred by position
Indirect triple	List of pointers to triples	Easy reordering

- **SSA (static single assignment)**: each variable is assigned only once.

7. Runtime Environment and Symbol Table

- **Activation record** (stack frame) holds: return value, actual parameters, control link, access link, saved registers, local data, temporaries.
- Storage: **static** (globals), **stack** (function calls), **heap** (dynamic memory).
- Parameter passing: call by value, reference, copy-restore, name.
- **Symbol table** stores names with type, scope, size, address. Usually built with a hash table.

8. Code Optimization and Code Generation

➤ Basic blocks and flow graphs

- **Basic block** = straight-line code with one entry and one exit.
- **Leaders**: first statement, target of any jump, statement just after a jump. Each leader starts a block.

Optimization techniques

Technique	Example
Constant folding	x = 2 * 3 becomes x = 6
Constant propagation	x = 5; y = x + 1 becomes y = 6
Common sub-expression elimination	Compute a + b once and reuse
Dead code elimination	Remove code whose result is never used
Copy propagation	After x = y, use y in place of x
Strength reduction	x * 2 becomes x + x or x << 1
Loop invariant code motion	Move unchanged computations outside the loop
Loop unrolling	Repeat the body to cut loop overhead

- **Peephole optimization**: look at a small window of target code: remove redundant loads / stores, unreachable code, jumps to jumps.
- **Data flow analysis**: liveness analysis, reaching definitions, available expressions.

> **EXAM POINT** A variable is **live** at a point if its current value is used later before being redefined. Live variables decide register allocation.

Code generation

- Issues: instruction selection, register allocation, evaluation order.
- **Register allocation** is done with graph colouring (interference graph).

9. Most Asked Questions (practice)

1. Explain the phases of a compiler with an example statement.
2. Count tokens in a given C code fragment.
3. Remove left recursion and apply left factoring on a grammar.
4. Find FIRST and FOLLOW sets and build the LL(1) table.
5. Compare LR(0), SLR, LALR and CLR parsers.
6. Construct the LR(0) item sets and SLR table for a small grammar.
7. S-attributed vs L-attributed definitions with examples.
8. Write three-address code, quadruples and triples for an expression.
9. Identify basic blocks and draw the flow graph.
10. Explain any five code optimization techniques.

TIP GATE compiler questions focus on token counting, FIRST/FOLLOW, LL(1) and LR conflicts, SDT evaluation, TAC with minimum temporaries, and liveness. Practise small grammars by hand.