Compiler Design

Short Notes based on ***Exit Exam***

Zemzem labs

1. Lexical Analysis (Lexer / Scanner)

Key Concepts

- Converts raw characters → **tokens**
  (Identifiers, keywords, literals, operators)
- Removes **whitespace & comments**.
- Reports **lexical errors** (illegal characters, malformed numbers).
- Maintains **line/column numbers** for error messages.

Important Notes

- **Token stream** is the output.
- Regular expressions define token patterns.
- Lexer may insert **symbol table entries** (e.g., for identifiers).

2. Syntax Analysis (Parsing)

Purpose

- Checks if token stream matches grammar.
- Constructs **parse tree** or **AST (Abstract Syntax Tree)**.

Parser Types

- **Top-down parsing:** LL(1), Recursive Descent
- **Bottom-up parsing:** LR, LALR, SLR

Important Notes

- **LR parsers** → more powerful, handle more grammars.
- **LL grammars** require:
  - No left recursion
  - Left factoring
- Operator **precedence & associativity** resolve ambiguous expression grammars.

Error Handling

- **Panic-mode**: skip tokens until synchronizing set.
- Must avoid "cascading errors".

3. Symbol Table

Purpose

- Stores information about identifiers:
    - Type
    - Scope level
    - Memory address
    - Storage class (local/global)
    - Function signatures

Notes

- Implemented using **hash tables** + **scope stack**.
- Enter scope → push table
- Exit scope → pop table

4. Semantic Analysis

Purpose

- Ensures the program is **meaningfully correct**.

Checks Include

- **Type checking** (e.g., int + float, function calls)
- **Scope resolution**
- **Arity** (number of arguments in function call)
- **Access control** (private, protected, public)
- Detecting:
    - Unused variables
    - Unreachable code

Attribute Grammars

- Attaches semantic rules to grammar productions.
- **Inherited attributes** flow down.
- **Synthesized attributes** flow up.

5. Intermediate Representation (IR)

Definitions

- A compiler **lower-level internal program representation** between source code and machine code.

Common IRs

- **Three-address code (3AC)** → statements like t1 = a + b
- **SSA (Static Single Assignment)** → each variable has one assignment
- **Bytecode** (JVM)
- **AST-based IR**

Role of IR

- Helps implement **machine-independent optimizations**.
- Makes back-end code generation easier.

6. Optimization Techniques

Optimization improves execution speed / reduces code size **without changing semantics**.

Local Optimizations

- **Peephole optimization**
  Replace inefficient instruction patterns.

Constant Folding

- Pre-compute constant expressions at compile time.

Dead Code Elimination

- Remove instructions whose results are unused.

Common Subexpression Elimination (CSE)

- Avoid recomputing repeated expressions.

Loop Optimizations

- **Loop invariant code motion (LICM)**
  Move invariants outside loop.
- **Loop unrolling**
  Replicate loop body to reduce branch overhead.

Interprocedural Optimizations

- Use data across functions:
  - Function inlining
  - Propagating constants across calls

Profile-Guided Optimization (PGO)

- Uses real runtime profile to optimize hot paths.

7. Code Generation

Role

- Map IR → **target machine instructions**.
- Choose efficient instructions (**instruction selection**).
- Assign registers (**register allocation**).

Key Notes

- **Graph-coloring register allocation** → optimal but slow.
- **Linear-scan allocation** → fast, used in JIT compilers.
- If registers insufficient → **spilling** occurs (store in memory).

8. Runtime Environment

Activation Records (Stack Frames)

Contain:

- Return address
- Saved registers
- Local variables
- Parameters

Calling Conventions

Rules controlling:

- Where arguments go (stack or registers)
- Who saves registers (caller vs callee)
- How return values are passed

9. Linking & Loading

Linking

- Combines object files.
- Resolves symbol references.
- Performs relocations (fix addresses).

Relocation

- Adjusting addresses of instructions/data when final layout is known.

Position-Independent Code (PIC)

- Code that can run regardless of load address.
- Used in **shared libraries (.so/.dll)**.

10. Garbage Collection & Memory

Garbage Collection

Automatic reclamation of memory no longer in use.

Common Techniques

- Mark & sweep
- Copying GC
- Generational GC

11. Advanced Compiler Concepts

Static Analysis

- Detects potential runtime errors at compile time.
- Examples: null-pointer checks, bounds checks.

Reaching Definitions

- Dataflow analysis identifying definitions that can reach a point.

Instruction Scheduling

- Reorder instructions to avoid pipeline stalls.

Ultra-Condensed Exam-Ready Summary

- Lexer → tokens
- Parser → AST
- Semantic → types/scopes
- IR → 3AC, SSA
- Optimization → CSE, DCE, LICM
- Code generation → instruction selection + register allocation
- Runtime → stack frames, calling conventions
- Linking → symbol resolution
- PIC → relocatable shared library code