**Unit I: Introduction to Translators and Lexical Analysis**:

**Translators, Compilers, and Interpreters**

A **translator** converts a program written in one language into another.

**1. Compiler**

* Translates the entire source code into machine code **before execution**.
* **Example:** C, C++ compilers.
* **Pros:** Faster execution.
* **Cons:** Debugging is harder since errors are detected after compilation.

**2. Interpreter**

* Translates and executes the code **line by line**.
* **Example:** Python, JavaScript interpreters.
* **Pros:** Easier debugging.
* **Cons:** Slower execution.

**3. Compilation Process**

The compilation process consists of **multiple phases**:

1. **Lexical Analysis:** Converts source code into tokens.
2. **Syntax Analysis:** Checks grammar and structure (Parsing).
3. **Semantic Analysis:** Ensures logical correctness.
4. **Intermediate Code Generation:** Converts to an intermediate representation.
5. **Optimization:** Improves code efficiency.
6. **Code Generation:** Converts to machine code.
7. **Code Linking & Loading:** Combines all modules for execution.

**Programming Language Grammars**

Defines the rules of a programming language.

**Derivations**

* **Leftmost derivation:** Expands the leftmost non-terminal first.
* **Rightmost derivation:** Expands the rightmost non-terminal first.

**Reductions**

* The process of replacing a group of symbols with a non-terminal using grammar rules.
* Used in **parsing** to simplify expressions.

**Regular Expressions**

* Define patterns for recognizing strings.
* Used in **lexical analysis** for token generation.
* Example: a\* (zero or more occurrences of 'a').

**Context-Free Language & Grammar (CFL & CFG)**

* **Context-Free Grammar (CFG):** Used to define programming language syntax.
* Consists of **terminals, non-terminals, start symbol and production rules**.

Example CFG for simple arithmetic:

E → E + T | T

T → T \* F | F

F → (E) | id

**Lexical Analyzer (Lexer)**

Converts source code into **tokens** (smallest meaningful units).

* Example: int a = 5; → Tokens: int, a, =, 5, ;

**Input Buffering**

* Used to improve efficiency by reading the input in **blocks** instead of character by character.

**Specification & Recognition of Tokens**

* Tokens are identified using **regular expressions** and matched using **finite automata**.

**Finite Automata**

Finite automata are mathematical models for recognizing patterns in text.

**Types of Finite Automata**

1. **NFA (Non-deterministic Finite Automata):**
   * Can have multiple transitions for the same input.
   * Easier to construct from regular expressions.
2. **DFA (Deterministic Finite Automata):**
   * Only one transition per input.
   * More efficient for lexical analysis.

**Regular Expressions to NFA**

* **Thompson’s construction** is used to convert **regular expressions** into **NFA**.

**Minimization of DFA**

* Reduces the number of states in DFA to make it more efficient.
* Done using **state equivalence and partitioning algorithms**.

**Keywords & Reserved Word Policies**

* **Keywords:** Predefined words in a language (if, while, return).
* **Reserved Words:** Words that cannot be used as variable names.

**LEX – Lexical Analyzer Generator**

* **LEX** is a tool used to generate lexical analyzers.
* Converts **regular expressions** into C code for token recognition.
* Works with **YACC (Yet Another Compiler Compiler)** for syntax analysis.

Example LEX Code:

%%

[0-9]+ printf("NUMBER");

[a-zA-Z]+ printf("IDENTIFIER");

%%

* Recognizes numbers and identifiers from input.

**Unit II: Syntax Analysis & Parsing**:

**Syntax Analyzer (Parser)**

* Checks whether a given source code follows the **grammatical rules** of a programming language.
* **Input:** Tokens from the lexical analyzer.
* **Output:** Parse tree or syntax tree.

**Context-Free Grammars (CFGs)**

A **Context-Free Grammar (CFG)** is used to define the syntax of programming languages.

* **CFG consists of:**
  + **Terminals (T):** Symbols that appear in the final output (if, else, +, -).
  + **Non-terminals (N):** Intermediate symbols used for derivation (E, T).
  + **Start Symbol (S):** The root of the grammar.
  + **Production Rules:** Define how non-terminals are replaced.

Example CFG for an arithmetic expression:

E → E + T | T

T → T \* F | F

F → (E) | id

**Top-Down Parsing**

* Begins parsing from the **start symbol** and derives the input string.
* **Works like recursive function calls.**

**Brute Force Parser**

* Also called **naïve parsing** or **exhaustive search parsing**.
* **Tries all possible derivations** until it finds the correct one.
* **Very inefficient** and not used in real-world compilers.

**Recursive Descent Parser**

* Uses **recursive functions** to expand grammar rules.
* **Manually written for each grammar rule.**
* Example:

void E() {

T();

if (lookahead == '+') {

match('+');

E();

}

}

* **Pros:** Simple to implement.
* **Cons:** Fails for **left-recursive grammars** (E → E + T).

**LL(1) Parser**

* **Left-to-right scanning**, **Leftmost derivation**, **1 lookahead symbol**.
* Uses **predictive parsing table** (non-recursive version of recursive descent).
* **Limitations:**
  + Cannot handle **left-recursion** (E → E + T).
  + Cannot handle ambiguous grammars.

**Bottom-Up Parsing**

* **Starts with the input string** and **reduces** it to the start symbol.
* Works like **shifting and reducing** tokens.

**Operator Precedence Parsing**

* **Uses operator precedence rules** to determine parsing.
* **Works well for arithmetic expressions** (+, -, \*, /).
* Example: 3 + 5 \* 2 → 3 + (5 \* 2) → (3 + 10) → 13

**Simple Precedence Parsing**

* Uses a **precedence table** to determine when to shift or reduce.
* **Example Precedence Table:**

+ > \*

\* > +

* + \* has **higher precedence** than +.

**LR Parsers (Bottom-Up Parsers)**

* **LR (Left-to-right, Rightmost derivation in reverse)**
* **More powerful than LL(1) parsers.**

**Types of LR Parsers**

1. **SLR (Simple LR) Parser:**
   * Uses a simple **parsing table**.
   * **Cannot handle all grammars.**
2. **LR (1) Parser:**
   * More powerful, **handles more complex grammars**.
   * Uses **lookahead (1 token) to decide reductions.**
3. **LALR (Look-Ahead LR) Parser:**
   * **Optimized version of LR(1)** with a smaller parsing table.
   * **Used in YACC and modern compilers.**

**YACC (Yet Another Compiler Compiler)**

* A tool to **generate parsers** automatically.
* Works with **LEX** to process input.

**Example YACC Code**

yacc

CopyEdit

%token NUM

%%

E : E '+' T | T ;

T : NUM ;

%%

* Defines a grammar for simple addition.

**Conclusion**

* **Top-down parsers** work by deriving the input from the **start symbol**.
* **Bottom-up parsers** reduce the input to the **start symbol**.

**Unit III: Syntax-Directed Translation**:

**1. Syntax-Directed Translation (SDT)**

* Syntax-Directed Translation is a method where **semantic actions** are associated with grammar rules.
* It helps in translating **high-level language constructs** into lower-level representations like intermediate code.
* It extends **context-free grammar (CFG)** by adding **semantic rules**.

**2. Implementation of Syntax-Directed Translators**

* A **Syntax-Directed Translator** consists of a **parser and an attribute evaluator**.
* The translator computes **attribute values** based on grammar rules and generates **intermediate representations** (e.g., syntax trees, intermediate code).

**3. Synthesized and Inherited Attributes**

Attributes are used to pass information in the **syntax tree**:

* **Synthesized Attributes:**
  + Computed from **children to parent** in the parse tree.
  + Example: Expression evaluation where parent value is computed from child values.
* **Inherited Attributes:**
  + Passed from **parent to child** nodes in the parse tree.
  + Example: Type checking in expressions where child nodes inherit the type of parent.

**4. Dependency Graph**

* A **graph representation** of attribute dependencies in a syntax tree.
* Nodes represent **grammar symbols**, and edges show **dependency relations**.
* Helps determine **evaluation order** of attributes.

**5. Evaluation Order of Attributes**

* **Top-Down Order (Preorder Traversal):** Used for **inherited attributes**.
* **Bottom-Up Order (Postorder Traversal):** Used for **synthesized attributes**.

**6. Construction of Syntax Trees**

* A **syntax tree** is a tree representation of the parsed program structure.
* Each **node represents an operation**, and **leaves represent operands**.

Example for 3 + 5 \* 2:

markdown

CopyEdit

+

/ \

3 \*

/ \

5 2

**7. Directed Acyclic Graph (DAG) for Expressions**

* A **DAG** is an optimized version of a syntax tree where **common subexpressions** are shared.
* Reduces **redundant computations** and helps in **code optimization**.

Example for a = b + c; d = b + c;

css

CopyEdit

+

/ \

b c

/ \

a d

* Here, **b + c** is computed **once** and used for both a and d.

**8. Bottom-Up Evaluation of S-Attributed Definitions**

* **S-Attributed Definitions:** Use only **synthesized attributes**.
* Evaluated in **postorder (bottom-up traversal)**.
* Example: **Expression evaluation** in arithmetic expressions.

**9. L-Attributed Definitions**

* **L-Attributed Definitions:** Use **both synthesized and inherited attributes**.
* Evaluated in **left-to-right order**.
* Example: **Type checking in declarations**.

**10. Top-Down Translation of L-Attributed Definitions**

* Attributes are **computed while parsing** in a **top-down manner**.
* Used in **recursive descent parsers** and **LL(1) parsing**.

**Conclusion**

* **Syntax-Directed Translation** associates **semantic rules** with grammar.
* **Synthesized & Inherited Attributes** help in **semantic analysis**.
* **Syntax Trees and DAGs** aid in **efficient expression evaluation**.
* **S-Attributed & L-Attributed Definitions** determine **evaluation strategies**.

**Unit IV: Errors, Intermediate Code, and Symbol Tables**:

**1. Errors in Compilation**

Errors occur at different phases of compilation and must be detected and handled properly.

**Lexical Phase Errors**

* Errors in **token recognition**.
* Example: Invalid characters in variable names (@var, 12abc).

**Syntactic Phase Errors**

* Errors in **grammar and structure**.
* Example: Missing semicolon (int a = 5) or unmatched parentheses.

**2. Intermediate Languages**

An **Intermediate Representation (IR)** is used between source code and machine code to make compilation more efficient.

**Postfix Notation (Reverse Polish Notation – RPN)**

* Operators follow operands.
* Example: A + B → A B +
* Removes the need for **parentheses** and follows **stack-based evaluation**.

**Syntax Tree**

* A hierarchical tree representation of expressions and statements.
* Example for a = b + c \* d:

=

/ \

a +

/ \

b \*

/ \

c d

**Parse Tree**

* Similar to a syntax tree but retains all grammar rules, making it more detailed.

**Three-Address Code (TAC)**

* A low-level representation where each statement has at most **three operands**.
* Example:

t1 = c \* d

t2 = b + t1

a = t2

**Triples**

* Represent TAC using **index-based references**.
* Example:

( \*, c, d ) → t1

( +, b, t1 ) → t2

( =, a, t2 )

**Indirect Triples**

* Similar to **triples**, but **stores references in a table** instead of direct indexing.

**3. Translation of Assignment Statements**

Converting a = b + c \* d; into TAC:

t1 = c \* d

t2 = b + t1

a = t2

This helps in **code optimization and efficient execution**.

**4. Symbol Tables**

A **symbol table** stores information about **variables, functions, and types** during compilation.

**Operations on Symbol Tables**

* **Insert:** Add new variables/functions.
* **Lookup:** Search for a variable’s details.
* **Update:** Modify existing symbols.
* **Delete:** Remove unused symbols.

**Symbol Table Organization for Non-Block Structured Languages**

* Uses **a single global table** for all symbols.
* Example: **BASIC, FORTRAN**

**Symbol Table Organization for Block-Structured Languages**

* Uses **nested symbol tables** for different scopes (functions, loops, etc.).
* Example: **C, Java, Python**

**Conclusion**

* **Error handling** is crucial in compilation.
* **Intermediate Representations** improve efficiency.
* **Symbol tables** help in managing variables and functions.
* **Three-address code, triples, and indirect triples** optimize code representation.

**Unit V: Run-Time Storage Management, Code Optimization, and Code Generation**:

**1. Run-Time Storage Management**

* Deals with how memory is allocated and managed during **program execution**.
* Includes **stack, heap, static, and code segments**.

**Storage Allocation and Referencing Data in Block-Structured Languages**

* **Static Allocation:** Memory assigned at compile-time (e.g., global variables).
* **Stack Allocation:** Used for **function calls** (local variables, recursion).
* **Heap Allocation:** Dynamic memory allocation (malloc, new in C/C++).

**2. Code Optimization**

Optimization improves execution time, memory usage, and efficiency.

**Sources of Optimization**

* **Common subexpression elimination** (x = a + b; y = a + b; → use t = a + b;).
* **Constant folding** (x = 5 \* 10; → replace with x = 50;).
* **Dead code elimination** (removing unused code).
* **Strength reduction** (x \* 2 → x << 1).

**Loop Optimization**

* **Loop invariant code motion:** Moves **constant expressions** out of the loop.
* **Loop unrolling:** Reduces iteration overhead.
* **Induction variable elimination:** Simplifies loop indexing.

**3. DAG (Directed Acyclic Graph) and Optimization of Basic Blocks**

* DAGs are used to **eliminate redundant computations** and optimize expressions.
* A **basic block** is a straight-line sequence of instructions **without jumps**.
* DAG helps in **common subexpression elimination** and **register reuse**.

Example DAG for x = a + b; y = a + b;:

+

/ \

a b

/ \

x y

**4. Code Generation**

Converts optimized intermediate representation into **machine code**.

**Machine Model**

* Defines how **instructions are executed** in a CPU.
* Includes **registers, memory access, and instruction set**.

**Next-Use Information and Register Allocation**

* **Next-Use Information:** Determines when a variable is used next to optimize register usage.
* **Register Allocation:** Assigns registers efficiently to **avoid excessive memory access**.

**A Simple Code Generator**

* Takes **three-address code (TAC)** as input and converts it into **machine code**.

Example TAC:

t1 = a + b

c = t1

Generated Assembly Code:

LOAD R1, a

ADD R1, b

STORE c, R1

**Code Generation from DAGs**

* Uses **DAGs to minimize redundant calculations**.
* Ensures **efficient use of registers**.

**5. Peephole Optimization**

A **local optimization technique** that improves small sequences of code.

**Common Techniques:**

* **Redundant load/store removal**: Removing unnecessary memory accesses.
* **Constant folding**: Precomputing values (x = 2 \* 5 → x = 10).
* **Algebraic simplifications**: Converting x \* 1 → x.

Example:  
Before:

LOAD R1, a

ADD R1, 0

STORE a, R1

After:

LOAD R1, a

(Removed ADD R1, 0 since it has no effect.)

**Conclusion**

* **Storage management** determines how memory is allocated and accessed.
* **Code optimization** removes inefficiencies for better performance.
* **Code generation** converts optimized code into machine instructions.
* **Peephole optimization** fine-tunes small sections of generated code.