Compiler Construction

Unit 1 - Introduction to Compiling

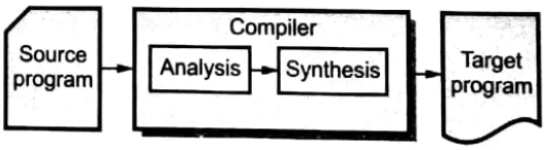
Q1. What is a compiler?

Answer: A compiler is a program that translates source code written in a high-level programming language into machine code or an intermediate language, which can be executed by a computer.

Q2. What is the Compiler Analysis-Synthesis Model?

Answer: The Compiler Analysis-Synthesis Model refers to the two main phases in a compiler:

1. Analysis: This phase reads the source code and breaks it down into its components, often generating intermediate representations. It consists of:
    - Lexical analysis (tokenizing the source code)
    - Syntax analysis (parsing the structure)
    - Semantic analysis (checking for semantic errors)
2. Synthesis: This phase generates the target code (machine code or intermediate code) from the analysis done in the previous phase. It includes:
    - Intermediate code generation
    - Code optimization
    - Code generation



In simple terms, analysis breaks down the input, and synthesis builds the output.

Q3. What are the properties a compiler should possess when built?

Answer:

1. Compiler should be bug-free
2. Correct machine code generation
3. Fast execution of generated code
4. Fast compilation
5. Portability
6. Good diagnostics and error messages
7. Compatibility with debuggers
8. Consistent optimization

Q4. Why should we study compilers?

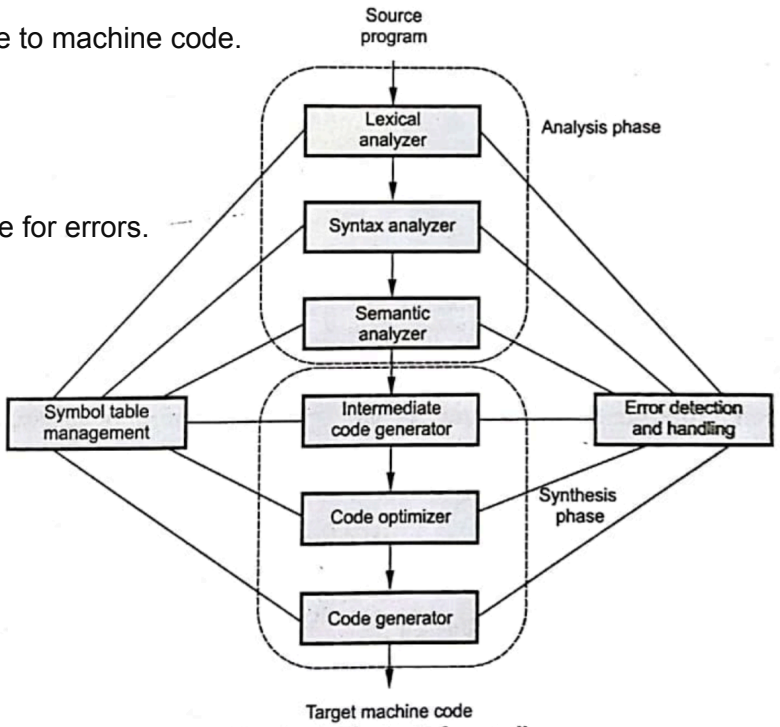
Answer:

1. Understand language implementation
2. Learn language design and syntax
3. Gain knowledge of program optimization
4. Learn hardware-software interaction
5. Improve problem-solving skills
6. Understand computer architecture
7. Learn development tools like debuggers

Q5. What are the phases of a compiler, and what tools are used in each phase?

Answer:
A compiler consists of several phases, each served by specific tools:

1. Lexical Analysis:
   - Task: Converts source code into tokens.
   - Tools: Lex/Flex.
2. Syntax Analysis:
   - Task: Checks if the token sequence follows language grammar.
   - Tools: Yacc/Bison.
3. Semantic Analysis:
   - Task: Ensures the program follows semantic rules (type checking, etc.).
   - Tools: Custom handling in the compiler.
4. Intermediate Code Generation:
   - Task: Converts parsed code into an intermediate representation (IR).
   - Tools: LLVM.
5. Optimization:
   - Task: Improves code efficiency.
   - Tools: LLVM.
6. Code Generation:
   - Task: Converts intermediate code to machine code.
   - Tools: GCC.
7. Debugging:
   - Task: Tests and inspects the code for errors.
   - Tools: GDB.



Q6. Elaborate all compilation phases with an example.

Answer:
The compilation process consists of several phases that transform source code into machine code. Using the example:

```
total = count + rate * 10;
```

Here are the phases explained in detail:

1. Lexical Analysis

In this phase, the source code is divided into tokens. For the given code, the tokens would be:

- total → Identifier
- = → Assignment Operator
- count → Identifier
- + → Addition Operator
- rate → Identifier
- * → Multiplication Operator
- 10 → Integer Literal
- ; → Semicolon (Punctuation)

2. Syntax Analysis

The tokens are arranged into a syntax tree that reflects the grammatical structure. For the given expression, the tree will represent operator precedence and how the operators interact with the operands:

```
      =
     / \
  total  +
        / \
   count  *
         / \
      rate  10
```

3. Semantic Analysis

This phase checks for logical errors in the code, ensuring that variables are declared and used correctly. It verifies:

- total, count, and rate are declared and compatible in terms of their types.
- Operations like rate * 10 and count + (rate * 10) are semantically correct.

4. Intermediate Code Generation

The syntax tree is converted into an intermediate representation, often using temporary variables. The intermediate code for this example could be:

```
t1 = rate * 10
total = count + t1
```

5. Code Optimization

The intermediate code is optimized to improve efficiency. In this case, there might be no further optimization, but generally, redundant calculations are removed or expressions are simplified.

6. Code Generation

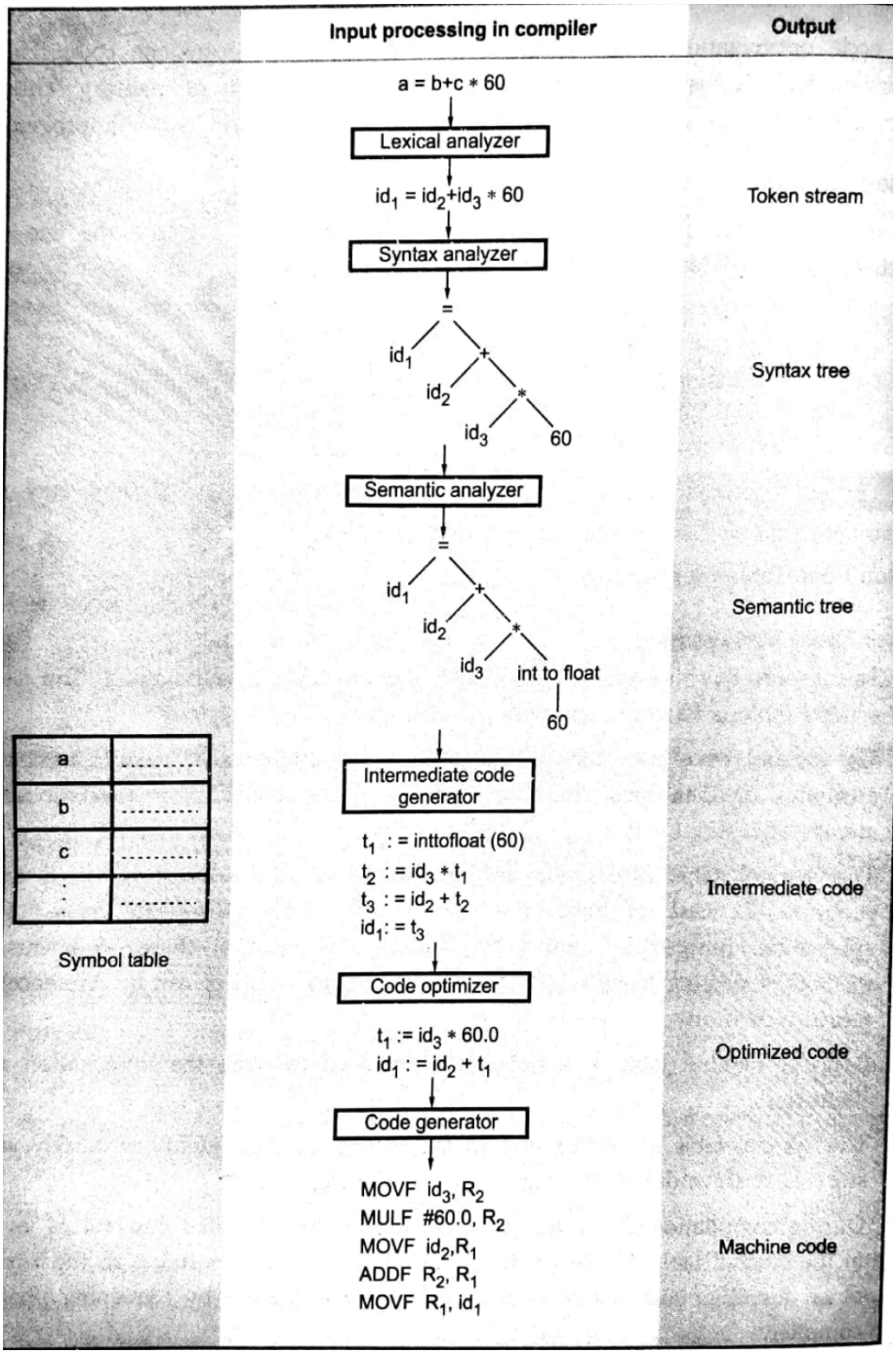
The intermediate code is converted into machine-specific assembly or machine code. An example in assembly for an x86 processor might be:

```
MOV AX, rate       ; Load rate into AX register
MUL 10             ; Multiply AX by 10
MOV BX, AX         ; Store result in BX (t1)
MOV AX, count      ; Load count into AX
ADD AX, BX         ; Add BX (rate * 10) to AX
MOV total, AX      ; Store result in total
```

Q7. Show how an input a = b + c * 10 gets processed in the compiler. Display the output at each stage of the compiler. Also, show the contents of the symbol table.

Answer:

Input:
a = b + c * 10;



Q8. What is grouping of phases?

Answer:
Grouping of phases in a compiler refers to organizing related phases into three main parts:

1. Front-End Phases: Focus on analyzing and translating the source code.
    - Lexical Analysis
    - Syntax Analysis
    - Semantic Analysis
2. Middle-End Phases: Handle intermediate code generation and optimization.
    - Intermediate Code Generation
    - Intermediate Code Optimization
3. Back-End Phases: Focus on generating target machine code and final optimizations.
    - Code Optimization
    - Code Generation
    - Code Linking and Assembly

Grouping phases helps streamline the compilation process and improve efficiency.

Q9. What is the concept of pass?

Answer:
A pass is a single traversal of code to perform a specific task.

- Single-pass compilers: Complete all phases in one pass.
- Multi-pass compilers: Use multiple passes for different phases.

Q10. What is a lexical analyzer? Explain in detail.

Answer:
A lexical analyzer (or lexer) is the first phase of a compiler that converts source code into tokens, which are the smallest units of meaningful data (keywords, operators, identifiers, etc.).

Function:
It reads the source code, removes whitespace and comments, and generates a stream of tokens for the next phase, syntax analysis.

Components:

- Input Buffer: Holds the source code.
- Scanner: Identifies tokens using patterns.
- Symbol Table: Stores identifiers and their attributes.

Example:

For a = b + 10;, the tokens are:

Token	Type
a	Identifier
=	Assignment Operator
b	Identifier
+	Addition Operator
10	Integer Literal
;	Semicolon

Q11. What are tokens, patterns, and lexemes?

Answer:

- Tokens: The smallest units of meaningful data identified during lexical analysis. They represent elements like keywords, operators, identifiers, literals, and punctuation.

Example: For the input a = 10 + b;, the tokens are:

Token	Type
a	Identifier
=	Assignment Operator
10	Integer Literal
+	Addition Operator
b	Identifier
;	Semicolon

- Pattern: A rule or description that defines the structure of a token. Patterns are often described using regular expressions or finite automata.

Example: A pattern for an identifier could be [a-zA-Z_][a-zA-Z0-9_]*, meaning it starts with a letter or underscore and can be followed by letters, digits, or underscores.

- Lexeme: The actual sequence of characters in the source code that matches a token's pattern.

Example: In a = 10 + b;, the lexemes are a, =, 10, +, b, and ;. These are the actual character sequences that the lexical analyzer matches to the token patterns.

Q12. What is input buffering? Explain one-buffer and two-buffer techniques.

Answer:

Input buffering is a technique that allows the lexical analyzer to read and process chunks of source code efficiently, minimizing I/O operations.

One-Buffer Technique:
In the one-buffer technique, a single buffer holds a chunk of the source code, and the lexical analyzer processes it sequentially. Once the buffer is processed, it is refilled with the next chunk.

Two-Buffer Technique:
In the two-buffer technique, while one buffer is being processed, the other is filled with the next chunk of input. This allows continuous processing without waiting for new data.

Q13. Why are space characters generally not allowed in identifier names?

Answer:
Space characters are not allowed in identifiers because they create ambiguity, splitting the name into multiple parts. Identifiers must be continuous to clearly represent a single entity.

Example: my variable would be seen as two separate identifiers: my and variable.

Q14. What are the specifications for recognition of tokens?

Answer:

The recognition of tokens is based on patterns, which are defined using regular expressions or finite automata. The specifications include:

1. Lexical Rules: Define patterns for different token types (keywords, operators, identifiers, etc.).
    - Example: An identifier can be defined as [a-zA-Z_][a-zA-Z0-9_]*.
2. Finite Automata: Used to recognize the patterns. The lexical analyzer uses a finite automaton to match the input characters to token patterns.
3. Symbol Table: Stores information about recognized identifiers (like variables or function names).

Q15. What is a regular expression?

Answer:
A regular expression (regex) is a pattern used to match and manipulate strings based on specific rules. It consists of literals and metacharacters.

Example:
The regex [a-z]+ matches one or more lowercase letters, where:

- [a-z] defines a character class for lowercase letters.
- + indicates one or more occurrences.

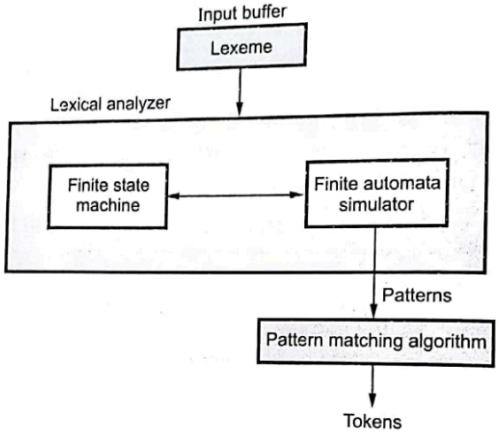
Regular expressions are used in lexical analysis to define patterns for tokens like identifiers, keywords, and numbers.

Q16. Explain the block schematic of lexical analyzer.

Answer:

The lexical analyzer recognizes tokens from the source code through the following steps:

1. Input Buffer: Source code is stored.
2. Lexeme: Lexemes are read from the buffer.
3. Regular Expressions: Patterns for tokens are created using regular expressions.
4. Finite Automata: A Non-deterministic Finite Automaton (NFA) is built from the regular expressions.
5. Finite Automata Simulator: The NFA is simulated to match lexemes with token patterns.
6. Pattern Matching Algorithm: The lexeme is matched with its token's pattern.
7. Tokens: Recognized tokens are produced.



Q17. What is the relationship between finite automata and regular expression?

Answer:

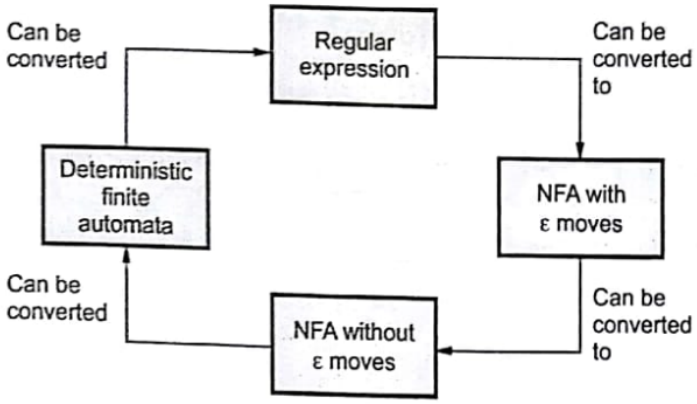
Finite automata and regular expressions are closely related as both define and recognize string patterns.

- Finite Automata: A model with states and transitions used to recognize patterns (DFA or NFA).
- Regular Expression: A sequence of characters that defines a pattern.

Relationship:

- Every regular expression can be converted into an NFA, and from an NFA, a DFA can be created.
- Both recognize the same set of patterns, known as regular languages.

In summary, finite automata implement the patterns described by regular expressions.



Q18. What are the finite automata implications?
Answer:
Finite automata are used to recognize regular languages and have the following implications:

1. Efficient Recognition: They provide efficient linear-time processing.
2. DFA vs NFA: DFAs are faster but larger, while NFAs are flexible but less efficient.
3. Conversion: An NFA can be converted to a DFA, though it may increase the number of states.
4. Memory Usage: DFAs generally use more memory than NFAs.

Overall, finite automata are crucial for recognizing patterns in lexical analysis.

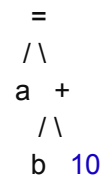
Unit 2 - Syntax Analysis

Q1. What is Syntax Analyzer or Parser, and what is its role?

Answer:

A Syntax Analyzer (or Parser) checks if the sequence of tokens produced by the lexical analyzer follows the grammatical structure of the language. It generates a parse tree or abstract syntax tree (AST) and detects syntax errors. The parser ensures that the input program adheres to the language's syntax rules, helping to validate the program's structure.

For the expression a = b + 10, the parse tree is:



The parser constructs this tree to represent the syntactic structure of the expression.

Q2. What are the basic issues in the parsing process?
Answer:

1. Ambiguity: A grammar can have multiple parse trees for the same input, leading to confusion.
2. Left Recursion: Rules that cause infinite recursion, breaking top-down parsers.
3. Error Handling: Parsers need mechanisms to detect and recover from syntax errors.
4. Efficiency: Complex grammars may lead to high memory and time usage.
5. Context-Sensitivity: Some languages have rules not easily captured by context-free grammars.
6. Lookahead: Parsers may require examining more than one token, complicating the process.

Q3. Why are lexical and syntax analyzers separated out?

Answer:
Lexical and syntax analyzers are separated to handle distinct tasks efficiently. The lexical analyzer generates tokens, while the syntax analyzer checks if the tokens follow the grammar. This separation allows for modular design, easier optimization, and improved efficiency.

Q4. What is CFG?

Answer:

A Context-Free Grammar (CFG) is a set of production rules used to define a language's syntax. Each rule has a single non-terminal on the left side and a mix of terminals and/or non-terminals on the right.

- Terminal: Basic symbols like keywords or operators.
- Non-terminal: Symbols that are expanded into terminals or other non-terminals.

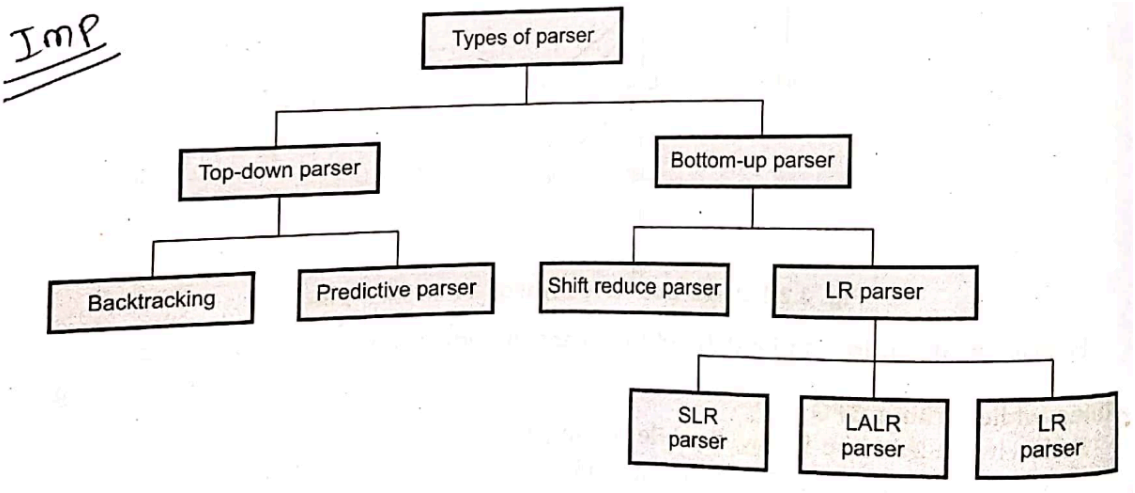
Example CFG:
S → aSb | ε

Diagram:

```
      S
    /   \
   a     S
       / \
      a   S
          |
          ε
```

In this example, S is a non-terminal, a and b are terminals, and ε represents the empty string.

Q5. What are the different Parsing techniques?



Syntax Analysis
Role of a Parser, Writing grammars for context free environments, Top-down parsing, Recursive descent and predictive parsers (LL), Bottom-Up parsing, Operator precedence parsing, LR, SLR and LALR parsers

Q6. Difference between Top-Down and Bottom-Up Parser

Answer:

Feature	Top-Down Parser	Bottom-Up Parser
Parsing Direction	Root to leaves	Leaves to root
Approach	Predictive (matches rules)	Reduces tokens to start symbol
Examples	Recursive Descent, LL Parser	Shift-Reduce, LR Parser
Left Recursion	Not handled well	Can handle left recursion
Error Detection	Early detection	Late detection
Efficiency	Simpler, less efficient	More efficient for complex grammars

Q7. Compare Recursive Descent and LL(1) Parsers

Answer:

Aspect	Recursive Descent	LL(1)
Definition	Manual, uses recursive functions.	Table-driven, top-down parser.
Backtracking	May backtrack without a lookahead.	No backtracking, uses 1-token lookahead.
Grammar Type	Handles more grammar, struggles with left recursion.	Restricted to LL(1) grammars.
Implementation	Manually written, harder to automate.	Automatable with parsing tools.
Performance	Slower with backtracking.	Faster, deterministic.

Q8. What is Top-Down Parsing? What are the Issues in Top-Down Parsing?

Answer:

Top-Down Parsing:
Top-down parsing starts from the start symbol of the grammar and tries to derive the input string by applying the production rules. It proceeds with a leftmost derivation, expanding the leftmost non-terminal at each step. This method tries to match the input with a sequence of non-terminal expansions.

Issues in Top-Down Parsing:

1. Backtracking: If the parser applies a rule and the input doesn't match, it must backtrack to try a different production, leading to inefficiency.

Example:
$A \rightarrow B \mid C$
$B \rightarrow x$
$C \rightarrow y$
For input y, the parser first tries $A \rightarrow B$ (fails) and then backtracks to try $A \rightarrow C$ (success).

Diagram:
$A \rightarrow B \rightarrow x$ (fails) $\leftarrow$ backtrack $\rightarrow A \rightarrow C \rightarrow y$ (success)

2. Left Recursion: Left recursion occurs when a non-terminal calls itself directly or indirectly in its own production, causing infinite recursion in top-down parsing.

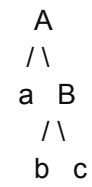
Example:
$A \rightarrow A\alpha \mid \beta$

Diagram:
$A \rightarrow A\alpha \rightarrow A\alpha\alpha \rightarrow A\alpha...$ (infinite loop)

3. Left Factoring: Left factoring is needed when multiple productions start with the same symbol. Without it, the parser can't determine which rule to apply based on the input.

Example:
A → ab | ac
Left factoring converts this to:
A → aB
B → b | c

Diagram:



4. Ambiguity: Ambiguity arises when a grammar allows multiple derivations for the same string, leading to confusion in parsing.

Example:
A → aA | bA | ε

Diagram:
For the string aa:

- First derivation: A → aA → aaA → aaε → aa
- Second derivation: A → aA → aε → a

Summary:

- Backtracking: Inefficient when multiple alternatives are available.
- Left Recursion: Leads to infinite recursion.
- Left Factoring: Resolves ambiguities when productions share a common prefix.
- Ambiguity: Leads to multiple valid derivations for the same string.

Q9. Demonstrate Left Recursion and Left Factoring with an Example

Answer:

Left Recursion:
Left recursion occurs when a grammar rule allows a non-terminal to appear as the leftmost symbol in its own production, directly or indirectly. This causes infinite recursion in top-down parsing.

Example:
Grammar with left recursion:
A → Aα | β
Here, A calls itself first (direct recursion).

Solution (Removing Left Recursion):
To eliminate left recursion, we rewrite the grammar by introducing a new non-terminal.
Rewritten Grammar:
A → βA'
A' → αA' | ε

Explanation:

- A produces β followed by zero or more occurrences of α.
- A' handles the repetition of α.

Diagram for Left Recursion (Before):

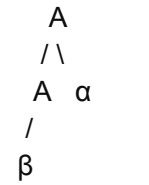


Diagram for Left Recursion (After Elimination):

Left Factoring:

Left factoring is required when multiple productions for a non-terminal share a common prefix, making it ambiguous for the parser to decide which production to use.

Example:

Grammar with common prefixes:
A → ab | ac
Here, both productions start with a, causing ambiguity.

Solution (Applying Left Factoring):

To remove the common prefix, factor it out into a separate production.
Rewritten Grammar:
A → aB
B → b | c

Explanation: A produces the common prefix a and delegates the choice (b or c) to B.

Diagram for Left Factoring (Before):

```
    A
   / \
  ab  ac
```

Diagram for Left Factoring (After Applying):

```
    A
   / \
  a   B
     / \
    b   c
```

Summary of Differences:

Left Recursion	Left Factoring
Causes infinite recursion in parsers.	Resolves ambiguities in common prefixes.
Eliminated by rewriting grammar with a new non-terminal.	Applied by factoring out the common prefix.

Summary:
Recursive Descent is simpler for small grammars, while LL(1) is efficient and automatable for larger ones.

Q10. Check if the Given Grammar is LL(1):
Grammar:
S → iCtS | iCtSes | a
C → b

Answer:

To check if a grammar is LL(1), the following conditions must be satisfied:

1. No Left Recursion:

The grammar does not have left recursion, so this condition is satisfied.

2. Left Factoring:

Factor common prefixes to avoid ambiguity.

- Productions for S share the prefix iCtS. After left factoring:

S → iCtSX | a
X → es | ε

3. First and Follow Sets:

Ensure First sets of alternatives are disjoint, and no overlap between First and Follow.

- First(S): {i, a}
- Follow(S): { $, es }
- First(X): {es, ε}

Observations:

- For S → iCtSX | a, First(iCtS) and First(a) are disjoint.
- For X → es | ε, First(es) and ε overlap in Follow(S) due to ε.

Conclusion:
The grammar is not LL(1) because there is an overlap between First(X) and Follow(S) caused by the nullable X.

Q11. Find Goto Values for the Given SLR Grammar

Given Grammar:
S → AB | gDa
A → ab | c
B → dC
C → gC | g
D → fD | g

Answer:
Steps to Compute Goto Values:

1. Augment the Grammar: Add a new start symbol S': S' → S
2. Construct the Canonical LR(0) Collection: Create sets of items (states) by closure and goto operations.
3. Goto Function: Goto transitions define state changes when the parser sees a specific grammar symbol (terminal or non-terminal).

State (I_i)	Items	Goto Transitions
I_0	S' → .S S → .AB S → .gDa	S → I_1, A → I_2, B → I_3, D → I_4
I_1	S' → S.	-
I_2	A → .ab, A → .c	a → I_5, c → I_6
I_3	B → .dC	d → I_7
I_4	D → .fD, D → .g	f → I_8, g → I_9
I_5	A → a.b	b → I_{10}
I_6	A → c.	-
I_7	B → d.C, C → .gC, C → .g	g → I_{11}
I_8	D → f.D	D → I_8
I_9	D → g.	-
I_{10}	A → ab.	-
I_{11}	**C → g.C, C → g.**	**g → I_{11}**

Summary of Goto Transitions:

- Goto(I_0, S) → I_1
- Goto(I_0, A) → I_2
- Goto(I_0, B) → I_3
- Goto(I_0, D) → I_4
- Goto(I_2, a) → I_5, Goto(I_2, c) → I_6
- Goto(I_3, d) → I_7
- Goto(I_4, f) → I_8, Goto(I_4, g) → I_9
- Goto(I_7, g) → I_{11}
- Goto(I_8, D) → I_8
- Goto(I_{11}, g) → I_{11}

Unit 3 - Syntax Derived Definitions

Q1. What are Syntax Directed Trees?

Answer:

Syntax Directed Trees (SDTs), or Abstract Syntax Trees (ASTs), represent the hierarchical structure of a program according to its grammar. They show the syntactic relationship between tokens and non-terminals, with each node representing a grammar symbol.

Key Points:

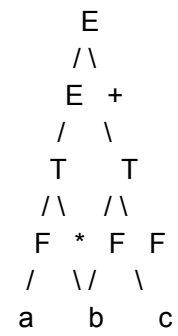
- Root: Start symbol of the grammar.
- Internal Nodes: Non-terminals.
- Leaf Nodes: Terminals (tokens).

Example:

For the expression a + b * c with the grammar:

E → E + T | T
T → T * F | F
F → a | b | c

The SDT is:



SDTs help in parsing, semantic analysis, and code generation.

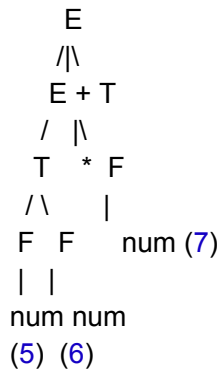
Q2. Write parse tree, syntax tree, annotated parse tree for 5*6+7 by assuming appropriate grammar rules.

Answer:

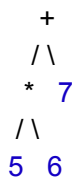
Grammar Rules (Assumed):

1. E → E + T | T
2. T → T * F | F
3. F → num

Parse Tree:



Syntax Tree:



Annotated Parse Tree:
Annotated trees add attributes like computed values (e.g., result of operations) to the parse tree. The attributes are typically added during syntax-directed translation.

Q3. Compare L-Attributes and S-Attributes

Feature	L-Attributed	S-Attributed
Definition	Attributes depend on parent or siblings (inherited).	Attributes are computed only from children.
Direction of Evaluation	Evaluated in a single left-to-right pass.	Evaluated bottom-up in post-order traversal.
Grammar Support	Works with both inherited and synthesized attributes	Works only with synthesized attributes.
Example	Inherited types in function declarations.	Expression evaluation in arithmetic trees.

Q4. Illustrate in brief about S-attributed definitions.

S-attributed definitions use synthesized attributes only, computed from child nodes. These definitions are common in bottom-up parsers.

Example:
For E → E1 + T, the value of E is synthesized as:
E.val = E1.val + T.val

Q5. Write parameter passing methods with examples.

1. Call by Value: Passes the value of the actual parameter. Modifications to the parameter don't affect the original value.
   Example: void foo(int x)
2. Call by Reference: Passes the address of the parameter, allowing the function to modify the original value.
   Example: void foo(int &x)
3. Copy-Restore: Combines call by value and reference; values are copied before and after function execution.
   Example: Used in Ada.
4. Call by Name: Actual parameters are substituted in the procedure body.
   Example: Used in ALGOL.

Q6. Why Every S-Attributed Definition is L-Attributed

Answer:

S-attributed definitions rely only on synthesized attributes, meaning no dependency on siblings or parent attributes. This inherently satisfies the conditions of L-attributed definitions.

Q7. Explain with an example bottom-up evaluation of inherited attributes.

Answer:

In bottom-up parsing, inherited attributes are computed by passing information from child nodes upwards.

Example:
For grammar:
S → A B
A → a { A.in = 1 }
B → b { B.in = A.in + 1 }

Input: ab

Steps:

1. Compute A.in = 1.
2. Use A.in to compute B.in = A.in + 1 = 2.

This evaluation ensures proper calculation of inherited attributes in a structured bottom-up manner.

Unit 4 - Run Time Environments

Q1. Discuss in brief any four source language related issues.

Answer:

1. Recursion:

If a language supports recursion, each recursive call requires memory for local variables and parameters. The number of active recursive instances is determined at runtime, affecting memory allocation.

2. Call by Value vs. Call by Reference:

Different memory allocation strategies are needed for passing parameters by value or by reference, influencing how memory is managed for function calls.

3. Non-local Name Access:

A procedure may need to access variables that are not local to it (e.g., global variables). The language must support methods for accessing non-local names.

4. Dynamic Memory Allocation:

If the language supports dynamic memory allocation and deallocation, memory is used more efficiently during runtime, as memory is allocated or freed as needed.

2. What is the sub-division of run-time memory and its significance?

Answer:

Run-time memory is divided into:

1. Target Code Area: Holds the generated target code, placed at the lower end of memory.
2. Data Objects Area: Stores data objects, allocated statically, placed above the code area (e.g., in FORTRAN).
3. Control Stack: Manages active procedures, storing procedure activations, local variables, and return addresses.
4. Heap Area: Used for dynamic memory allocation during execution, with memory for data items and activations.

The stack and heap sizes are flexible and can grow or shrink during execution.

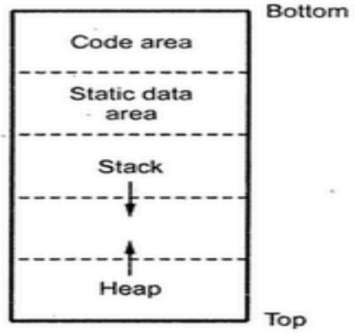


Fig. 6.1 Run-time storage organization

Q.3 Compare all storage allocation strategies (Static, Stack, and Heap Allocation).
Answer:

Feature	Static Allocation	Stack Allocation	Heap Allocation
Time of Allocation	Done at compile time.	Done at runtime, using the stack.	Done at runtime, dynamically using heap.
Memory Allocation	No dynamic data structures. Data size is fixed.	Supports dynamic data structures but with LIFO order.	Supports dynamic data structures and objects.
Support for Recursion	Memory bound to data objects at compile time.	Memory allocated in a LIFO order, pushed onto the stack.	Memory allocated from a contiguous block in the heap.
Efficiency	Does not support recursion.	Supports recursion using stack frames.	Supports recursion but slower due to dynamic memory allocation.
Memory Reuse	Simple and efficient, but cannot handle dynamic memory.	Efficient but slower than static allocation for large data.	Efficient in memory management but may cause fragmentation.
Limitations	No memory reuse, fixed allocation. Does not support recursion or dynamic data structures.	Memory is automatically reclaimed when activation frames end. Does not retain non-local variables after stack unwinding.	Memory can be reused, but fragmentation may occur. Can lead to fragmentation if not managed well.

Q.4 Explain in brief any two parameter passing methods.

Answer:

Method	Call by Value	Call by Reference
Description	The actual parameters are evaluated, and their values are passed to the called procedure. The formal parameters can be changed, but the actual ones remain unchanged.	The address of the actual parameters is passed to the called procedure. Changes to the formal parameters will affect the actual parameters.
Example	C, C++ (non-reference parameters)	C++ (reference parameters), Pascal (var parameters)

Q5. Define various approaches to symbol table organizations.

Answer:
A symbol table is a data structure used by a compiler to store information about variables, constants, functions, and other identifiers. It helps in tracking scope and binding information. The main methods for organizing symbol tables are:

1. List Data Structure:
   - Stores names in a linear list (array).
   - Simple and minimal space but slow search and prone to errors like “undeclared name.”
2. Self-Organizing List:
   - Uses a linked list where frequently accessed names are moved to the front.
   - Faster access for popular names but still slow for others.
3. Hash Tables:
   - Uses a hash function to map names to table locations.
   - Fast search but complex to implement and requires extra memory. Collision handling needed.

These methods balance speed and memory usage for symbol table management.

Unit 5 - Intermediate Code Generation

Q1. What is intermediate code in compiler design, and what are its benefits and properties?

Answer:
Intermediate code is a machine-independent representation of the source program used in the compilation process. It serves as a bridge between the front end (syntax analysis) and the back end (code generation).

Benefits:

1. Machine Independence: Allows compilers for different machines with the same front end.
2. Language Independence: Supports multiple source languages on the same machine.
3. Optimization: Enables machine-independent optimization.

Properties:

1. Efficient to Generate: Easily produced by the compiler.
2. Facilitates Code Generation: Helps in efficient final code generation.
3. Flexible for Optimization: Supports optimization before final code generation.

Q2. Write comparison about quadruple, triple, and indirect triple.

Answer:

1. Quadruple:
   - Representation: (Op, arg1, arg2, result)
   - Example: (-, a, b, t1)
   - Advantage: Clear separation of operation, operands, and result.
   - Disadvantage: Requires extra space to store the result.
2. Triple:
   - Representation: (Op, arg1, arg2)
   - Example: (-, a, b)
   - Advantage: More compact than quadruples.
   - Disadvantage: Requires indirect addressing to retrieve results.

3. Indirect Triple:
    - Representation: (Op, arg1, arg2) with references to result locations
    - Example: (-, a, b)
    - Advantage: Further compact than triples and avoids redundant result storage.
    - Disadvantage: Slightly more complex to handle result addresses.

Q3. Write the following expression in all types of intermediate representations you know: (a-b) * (c+d) - (a+b)

a) Three-Address Code (TAC):
t1 = a - b
t2 = c + d
t3 = t1 * t2
t4 = a + b
result = t3 - t4

b) Quadruples:
(Op, arg1, arg2, result)
1. (-, a, b, t1)
2. (+, c, d, t2)
3. (*, t1, t2, t3)
4. (+, a, b, t4)
5. (-, t3, t4, result)

c) Triples:
(Op, arg1, arg2)
1. (-, a, b)
2. (+, c, d)
3. (*, 1, 2)
4. (+, a, b)
5. (-, 3, 4)

d) Indirect Triples:
(Op, arg1, arg2)
1. (-, a, b)
2. (+, c, d)
3. (*, result_of_1, result_of_2)
4. (+, a, b)
5. (-, result_of_3, result_of_4)

Q4. Write about case statements in case of intermediate code generation, with appropriate grammar rules and related semantic actions and with related examples too.

Answer:

Grammar Rules for Case Statements:
stmt → CASE expr : stmt1
stmt → CASE expr : stmt1 | CASE expr : stmt2
stmt → CASE expr : stmt1 | CASE expr : stmt2 | DEFAULT : stmt3

Semantic Actions:

- Evaluate the expression in the CASE statement.
- Compare the expression with the cases.
- If a match is found, jump to the corresponding label.
- If no case matches, jump to the DEFAULT label.

Example:

For the expression switch (x) { case 1: stmt1; case 2: stmt2; default: stmt3; }
Intermediate Representation:

```
t1 = x
if t1 == 1 goto label1
if t1 == 2 goto label2
goto label_default
label1: stmt1
goto end
label2: stmt2
goto end
label_default: stmt3
end:
```

Q5. Write any four language constructs for intermediate code forms.

Answer:

1. Assignment Statements:
   - Example: a = b + c
   - Represents the assignment of the result of an operation to a variable.
2. Conditional Statements:
   - Example: if (x > y) goto label
   - Represents a decision based on a condition.
3. Loops:
   - Example: while (i < 10) { i = i + 1; }
   - Represents an iterative construct.
4. Function Calls:
   - Example: call foo(a, b)
   - Represents invoking a procedure or function.

Q6. Demonstrate intermediate code generation for declarations.

Answer:

For variable declarations such as:
int a;
float b;

Intermediate Code:
t1 = allocate int
t2 = allocate float
This allocates memory for integer a and floating-point b.

Q7. Discuss benefits and properties of intermediate code.

Answer:

1. Portability: Intermediate code provides a machine-independent representation, making it portable across different hardware platforms.
2. Optimization: Allows optimization techniques like constant folding and dead code elimination.
3. Separation of Concerns: Separates syntax analysis (front end) from code generation (back end), allowing for easier maintenance and debugging.
4. Simplifies Code Generation: Since it abstracts away machine-specific details, it simplifies the translation process to machine code.

Q8. Write details about procedure calls in case of intermediate code generation, with appropriate grammar rules and their related semantic actions and related examples too.

Answer:

Grammar Rules for Procedure Calls:
stmt → CALL procedure-name (arg-list)

Semantic Actions:

1. Evaluate arguments.
2. Transfer control to the procedure.
3. After execution, return the control to the calling point.

Example:
For the procedure call call foo(a, b), intermediate code generation:

Intermediate Representation:
t1 = a
t2 = b
call foo, t1, t2
This means we pass the values of a and b to the procedure foo.

Unit 6 & 7 - Code Generation, Optimization and Advanced Compilation Techniques

Q1. What is runtime storage management in code generation? Explain briefly.

Answer: Runtime storage management handles memory during program execution, including:

- Memory Allocation: Assigning space for variables (local, global).
- Scope Management: Managing variable visibility (local/global).
- Activation Records: Storing return addresses and local variables during function calls.
- Dynamic Memory: Allocating and deallocating memory during execution.

Example: During a function call, the compiler allocates space on the stack for the function's local variables and parameters.

Q2. What is runtime storage management in code generation? Explain briefly.

Answer: Runtime storage management handles memory during program execution, including:

- Memory Allocation: Assigning space for variables (local, global).
- Scope Management: Managing variable visibility (local/global).
- Activation Records: Storing return addresses and local variables during function calls.
- Dynamic Memory: Allocating and deallocating memory during execution.

Example: During a function call, the compiler allocates space on the stack for the function's local variables and parameters.

Q3. Define basic blocks, flow graphs with relevant examples.

Answer:

- Basic Blocks: A sequence of consecutive statements with no internal jumps. It has one entry and one exit point.

Example:

```
a = b + c   // Block 1
if a > 0    // Block 2
b = a + d   // Block 3
```

- Flow Graph: A graph where nodes represent basic blocks, and edges represent control flow between them.

Example:

(start) --> [BB1] --> [BB2] --> [BB3] --> (end)

Q4. Demonstrate four loop optimization techniques and illustrate five concepts of code generation.

Answer:

1. Loop Unrolling: Reduces loop iterations by performing multiple operations in one cycle.
2. Loop Fusion: Combines adjacent loops to reduce overhead.
3. Loop Inversion: Changes loop direction for better memory access.
4. Strength Reduction: Replaces expensive operations with cheaper alternatives.

Q5. Algorithm for Partitioning into blocks and illustrating Peephole optimization. Partitioning Algorithm:

Answer:

1. Start from the first instruction.
2. Mark entry points for basic blocks.
3. Split blocks at jumps/branches.
4. Repeat for all instructions.

Peephole Optimization:

Replaces inefficient instruction patterns with better alternatives.

Example:

Before:

```
a = b + c
t1 = a + d
result = t1
```

After:

```
a = b + c
result = a + d
```

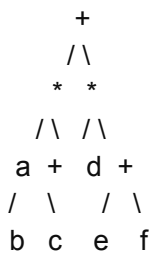
Q6. Explain code generation from DAG through algorithm, its process, and through the example also.

Answer:
DAG (Directed Acyclic Graph) helps break down expressions and reuses common parts to avoid redundant calculations.
Algorithm:

1. Build the DAG: Create a node for every operation and operand.
2. Traverse DAG: Start from leaves to root, generating code for each operation.
3. Reuse Sub-expressions: Reuse results for repeated calculations.

Example:
Expression: a * (b + c) + d * (e + f)

1. DAG:



2. Code:

```
t1 = b + c
t2 = a * t1
t3 = e + f
t4 = d * t3
result = t2 + t4
```

Here, b + c and e + f are computed once and reused.

Q7. Explain data flow analysis and equations in detail with relevant examples.

Answer:

Data Flow Analysis tracks how data moves through the program, helping identify unused code or optimize variables.

Types of Data Flow:

1. Reaching Definitions: Which variable definitions reach a particular point.
2. Live Variables: Which variables will be used later in the program.
3. Available Expressions: Which expressions are already calculated and can be reused.

Equations:

- Reaching Definitions:

  OUT[B] = IN[B] - KILL[B] + GEN[B]

  - KILL[B]: Definitions overwritten in block B.
  - GEN[B]: New definitions made in block B.
- Live Variables:

  OUT[B] = IN[B] - KILL[B] + GEN[B]

Example:

For:
a = b + c;
x = a + d;

- Reaching Definitions: Tracks when a is defined and where it's used.
- Live Variables: a is live at x = a + d; because it's used later.

This helps remove unused code or optimize calculations.