GeeksforGeeks

GATE फर्रे CSE

COMPILER DESIGN

SHORT NOTES

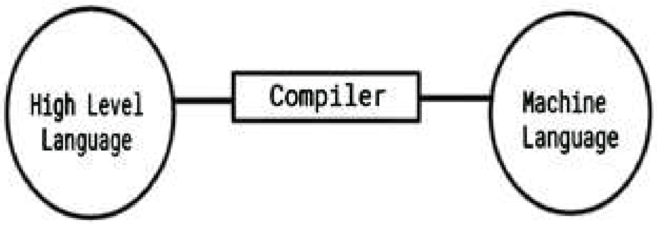
ENROLL NOW

TO EXCEL IN GATE
AND ACHIEVE YOUR DREAM IIT OR PSU!

ENROLL NOW

1. INTRODUCTION TO COMPILER

- A compiler is a type of translator that converts a program written in a high-level language (called source language) into a low-level language (called machine or object language).



- If there is any mistake in the code, the compiler gives a compilation error.
- High-level languages can perform more than one operation in a single statement. These languages are easy to understand and use, like C, C++, Java, or Python.
- Low-level languages can perform at most one operation in a single statement. These are close to the machine and difficult for humans to understand, like Assembly or Machine Code.
- A translator is a general term for any tool that converts code from one language to another.
- A compiler is a specific kind of translator that works from high-level to low-level language.

There are two main parts of a compiler:

1. **Analysis Phase**
   In this phase, the source code is checked and an intermediate representation is created. It includes:

○ **Lexical Analyzer**
→ Breaks the source code into tokens
→ Uses DFA (Deterministic Finite Automaton)
→ Checks spelling-like errors (example: wrong keywords or characters)

○ **Syntax Analyzer**
→ Checks grammatical structure of the code
→ Uses a parser (Parser is a DPDA – Deterministic Pushdown Automaton)
→ Example: Missing semicolon, incorrect nesting

○ **Semantic Analyzer**
→ Checks the meaning of the program
→ Ensures correct use of variables, functions, etc.
→ Example: Type mismatch, stack overflow

2. **Synthesis Phase**
   In this phase, the final output (target code) is generated from the intermediate code. It includes:

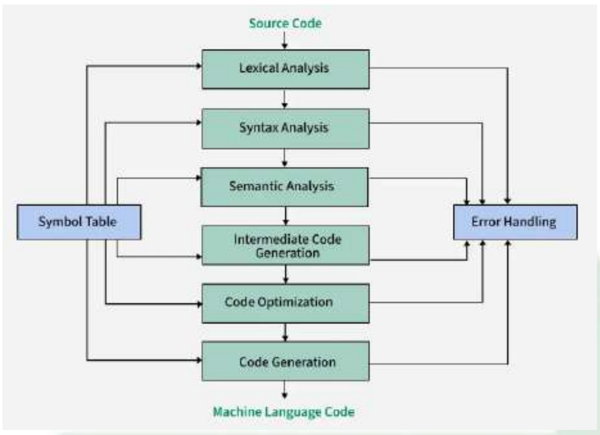
○ **Intermediate Code Generator**
→ Converts high-level code into an easy-to-optimize intermediate form
→ Helps in reusability and portability

○ **Code Generator**
→ Converts intermediate code into low-level target code (machine code)

○ **Code Optimizer**
→ Improves performance of code without changing output
→ Techniques:
(i) Loop invariant code motion
(ii) Common subexpression elimination
(iii) Strength reduction
(iv) Function inlining
(v) Dead code elimination

Symbol Table

- Stores information about variables, functions, objects etc.
- It is a data structure shared by all phases of the compiler
- Meta-data: data about the data used in program



Language Processing System :

When we write a program in High-Level Language (HLL), it goes through multiple steps before becoming executable:

1. **Preprocessor**:
   Takes input HLL code and removes comments, expands macros, and includes header files. Output is Pure HLL.

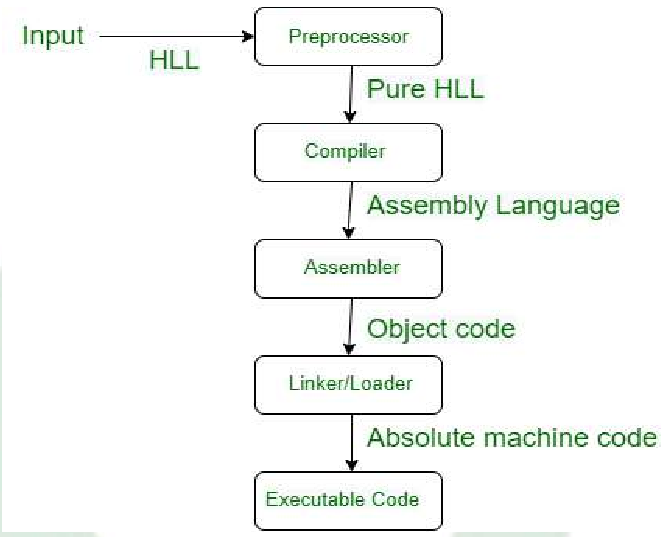
2. **Compiler**:
   Converts Pure HLL into **Assembly Language** (which is low-level but human-readable).

3. **Assembler**:
   Converts Assembly Language into **Object Code** (machine-readable but incomplete).

4. **Linker/Loader**:
   Links object code with other files (like libraries) and converts it into **Absolute Machine Code**.

5. **Executable Code**:
   Final code which runs on the machine.

So the final output is the executable file that we can run.



2. LEXICAL ANALYSIS

Introduction to Lexical Analysis

➢ Lexical Analysis is the **first phase of a compiler**.

➢ It reads the source code **character by character** and converts it into **tokens**.

➢ The lexical analyzer is also called **scanner** or **token recognizer**.

➢ It also detects **lexical errors**, such as invalid characters.

Functions of Lexical Analyzer

➢ Read the entire program character by character.

➢ Converts input into a **sequence of tokens**.

➢ Removes **comments**, **whitespace**, and **tabs**.

➢ Applies the **Maximal Munch Rule** (selects the longest possible valid token).

➢ Uses **regular expressions** to define token patterns.

➢ Uses **Finite Automata** (usually DFA) to implement token recognition.

➢ Passes tokens to the **syntax analyzer** along with their **attributes**.

Important Definitions

➢ **Lexeme**: A sequence of characters in source code that matches a token pattern.
Example: x, +, 123

➢ **Token**: A structured representation of a lexeme, usually as a pair:
<token-type, attribute>.
Example: <IDENTIFIER, name>, <NUMBER, value>

➢ **Attribute**: Extra information associated with the token, such as the actual name of a variable or value of a constant.

How to calculate number of tokens:

➢ To count tokens, break the line of code into small parts like keywords, identifiers, operators, literals, and punctuation symbols.
➢ Each part is counted as one token.

Common types of tokens:

- Keywords – like int, if, while
- Identifiers – variable or function names like sum, main
- Operators – like +, -, =, *, /
- Literals – values like 5, "hello"
- Punctuation/Special Symbols – like ;, (), {}

Example: **int sum = a + 5;**
Tokens in this line:
1. int, 2. sum, 3. =, 4. a, 5. +, 6. 5, 7. ;

Total tokens = 7

Note: Spaces are not counted as tokens.

3. Syntax Analysis

- ➢ **Syntax analyzer** is also known as **parser**.
- ➢ The **syntax** (structure) of a programming language is defined using **context-free grammar (CFG)**.
- ➢ A **parser** takes the **stream of tokens** (which are generated by the lexical analyzer) and checks whether they follow the correct syntax of the language or not.
- ➢ Grammatical errors in a program are checked using **parsers**.
- ➢ Parsers are basically **DPDA** (Deterministic Pushdown Automata)
- ➢ All parsers are **table-driven**, meaning they use parsing tables to decide how to process input.

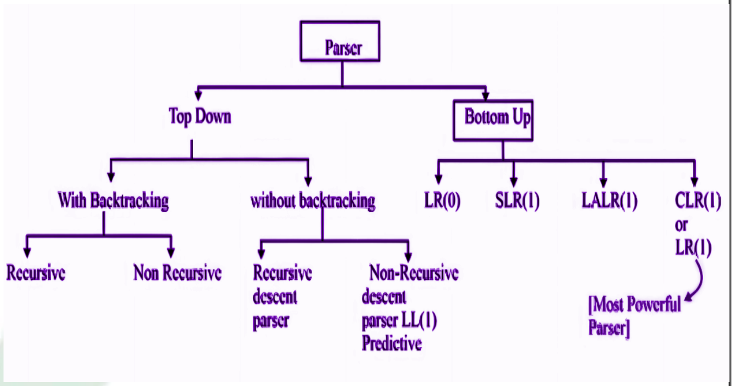
Parsers are mainly of two types:

1. **Top-Down Parsing**

   - ○ Starts from the root of the parse tree and goes towards the leaves.
   - ○ It can be of two types:
   - ■ **With backtracking**
   - ■ Example: Recursive Parser
   - ■ **Without backtracking**
   - ■ Examples:
   - ■ Recursive Descent Parser
   - ■ Non-Recursive Descent Parser (**LL(1)** Parser / Predictive Parser)

2. **Bottom-Up Parsing**

   - ○ Starts from leaves and moves towards the root of the parse tree.
   - ○ Types of Bottom-Up Parsers:
   - ■ **LR(0)**
   - ■ **SLR(1)**
   - ■ **LALR(1)**
   - ■ **CLR(1)** or **LR(1)** → *This is the most powerful parser*



Ambiguous Grammar:

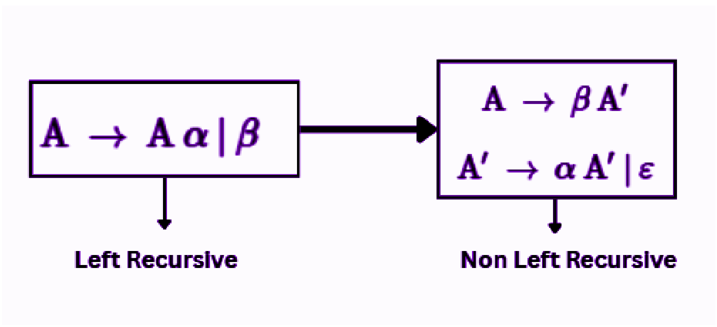
If a single input string has more than one parse tree, then the grammar is called ambiguous grammar.

Left Recursion:

- A grammar like $A \rightarrow A\alpha \mid \beta$ is called left recursive.
- Top-down parsers cannot handle left recursion, so we convert grammar into right recursion.

Left Recursion Elimination:

- To remove left recursion, we do the following:


$A \rightarrow A\alpha \mid \beta$
$A \rightarrow \beta A'$
$A' \rightarrow \alpha A' \mid \varepsilon$
Left Recursive
Non Left Recursive

Problem: Left recursive grammar can lead to infinite loop in top-down parsers that use Left Most Derivation.

Left Factoring:

- A predictive parser (top-down parser without backtracking) needs the grammar to be left factored.
- Left factoring removes common prefixes and helps avoid backtracking.
- It does not remove ambiguity but helps the parser to choose the correct rule easily.

If any grammar has common prefixes like:

$A \to \alpha\beta_1 \mid \alpha\beta_2 \mid \ldots \mid \alpha\beta_n \mid \gamma$

We convert it to:

$A \to \alpha A' \mid \gamma$

$A \to \beta_1 \mid \beta_2 \mid \ldots \mid \beta n$

This is called **left-factored** grammar.

Top-Down Parsing (TDP)

➢ **Recursive-Descent Parsing**:

- A parsing technique that follows the brute force method.
- It suffers from backtracking, which makes it inefficient. It is not widely used and is more general-purpose.
- **Predictive Parsing**:This method chooses the correct production rule by simply looking at the current input symbol.
- **Types of Predictive Parsing**:
  - **Recursive Predictive Parsing**: Each non-terminal symbol is represented by a separate procedure.
  - **Non-Recursive Predictive Parsing**: A table-driven approach, also known as **LL(1) Parsing**.

LL(1) Parser / Predictive Parser

- The **LL(1) grammar** is unambiguous, left-factored, and non-left recursive.
- The parser follows a **left-most derivation** approach.
- **LL(1)**: Refers to the number of look-ahead symbols being **1.**
- It uses **1 Look-Ahead Symbol** for deciding the next production.
- **Left-Most Derivation**: Parsing starts from the left-most symbol in the production rule.
- **Left to Right Scanning**: The input string is scanned from left to right.

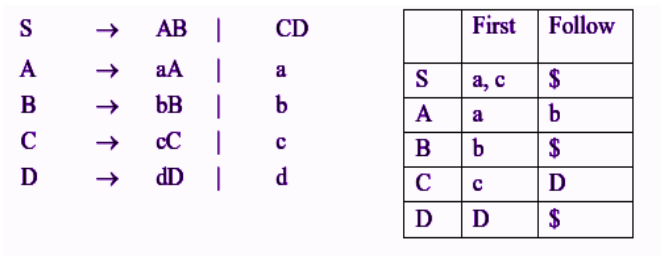
First Set

- The First Set represents the **extreme left terminal** from which the string of a variable starts.
  - It never contains a variable but **can contain** the **empty string (ε)**.
  - The **First Set** can always be determined for any variable.

Follow Set:

- The Follow Set contains **terminals** and the **special symbol $** (end of input).
- It **never contains** variables or the **empty string (ε)**.
- **How to Find the Follow Set**:
  1. **Include $** in the **follow set** of the **start variable**.
  2. **If the production is of the form**: $A \to \alpha B\beta$
     - **Follow(B)** = **First(β)**.
     - If **β** → **ε** (epsilon, empty string), then **Follow(B)** = **Follow(A)**.
  3. **For productions like**: $A \to \alpha A$ (where a variable repeats in the production), there is **no Follow Set** for A.

Example of First and Folow Set



$$
\begin{aligned}
S &\rightarrow AB \mid CD\\
A &\rightarrow aA \mid a\\
B &\rightarrow bB \mid b\\
C &\rightarrow cC \mid c\\
D &\rightarrow dD \mid d
\end{aligned}
$$

	First	Follow
S	a, c	\$
A	a	b
B	b	\$
C	c	D
D	D	\$

Algorithm to Construct Parsing Table:

Remove Left Recursion (if any).

1. **Left Factor** (remove common prefixes in productions).
2. **Find First and Follow Sets**.
3. **Construct the Parsing Table** using First and Follow sets.
4. **Increase Look-Ahead Symbol** if necessary for better accuracy.

LL(1) Parsing Table Construction

1. Use **First** and **Follow** sets to build the table.
2. For each production **A → α**

    a. If a terminal **'a'** is in **First(α)** → add **A → α** in table at **[A, a]**.

    b. If **First(α)** has **ε** → for each **'b'** in **Follow(A)**, add **A → ε** in **[A, b]**.

LL(1) Grammar Condition Checking:

To check if grammar is **LL(1):**

1. If **$A \rightarrow \alpha_1 \mid \alpha_2$**, then:
   $\text{First}(\alpha_1) \cap \text{First}(\alpha_2) = \emptyset$

2. If **$A \rightarrow \alpha \mid \varepsilon$**, then:
   $\text{First}(\alpha) \cap \text{Follow}(A) = \emptyset$

Exam Focus

- A grammar is **LL(1)** if its parsing table has **no multiple entries**.
- **Left recursive**, **not left factored**, or **ambiguous grammars cannot be LL(1)**.

Bottom-Up Parsing (Shift Reduce Parsing)

Introduction to Bottom-Up Parsing

➢ Bottom-up parsing is the method of building the **parse tree from input to start symbol** of grammar.

➢ It works by **reducing** the input string step by step using grammar rules.

➢ It follows the **reverse of rightmost derivation (RMD)**.

NOTE:

- **Handle**: A part of the input string that matches the **right-hand side (RHS)** of a production.
- **Handle Pruning**: The process of identifying the handle and replacing it with the **left-hand side (LHS)** of the corresponding production.

Characteristics of Bottom-Up Parsing

★ Also called as **shift-reduce parser**.

★ Can be used for **ambiguous and unambiguous grammars**.

★ It **simulates the reverse of rightmost derivation**.

★ **More powerful** than top-down parsing techniques.

★ Time complexity: **$O(n^3)$** in the general case.

★ **Handle pruning** is the main source of overhead.

★ Capable of handling **left recursion** and **common prefixes**.

★ Some **unambiguous grammars** may still not have any bottom-up parser.

Types of Bottom-Up Parsers

Bottom-up parsers include various **LR-based parsers**, such as:

1. **LR(0)**
2. **SLR(1)** – Simple LR
3. **LR(1)** – Canonical LR
4. **LALR(1)** – Look-Ahead LR
5. **CLR(1)** – Canonical LR (same as LR(1))

Note:
➔ **LR(k)** parser:

◆ **L** → Scans input from **Left to right**

◆ **R** → Produces **Reverse of Rightmost Derivation**

◆ **k** → Number of **Lookahead symbols**

➔ **LR(1)** = LR(0) + 1 Lookahead symbol

LR(0) Parser:

- **LR(0) Parser** is used to analyze **LR(0) grammars**. It's designed to avoid conflicts in the parsing table, meaning no multiple actions are allowed for the same state.

Conflicts in LR(0) Parser:

1. Shift-Reduce (SR) Conflict:

- This happens when, in one state, both **shift** and **reduce** actions are possible.
    - **Shift** means moving to the next symbol.
    - **Reduce** means applying a rule to reduce part of the string.
- The parser doesn't know whether to **shift** or **reduce**, causing a **conflict**.

Example:

- If you have **A → α·xβ** (shifting) and **B → γ·** (reducing), then parser can't decide what to do.

2. Reduce-Reduce (RR) Conflict:

- This happens when **two reduce** actions are possible in the same state.
- The parser doesn't know which reduction to apply.

Example:

- If both **A → α·** (reducing) and **B → γ·** (reducing) are possible in the same state then its RR conflict.

Closure and Goto Functions:

- **Closure()** and **Goto()** are used to build the **LR items**.
    - **Closure() adds** more items to a state, expanding it.
    - **Goto() moves** from one state to another, following transitions.

LR(0) Parsing Table Construction:

While creating the **parsing table**:

1. **If GOTO (I_k, a) = I_j**, set the action as **shift**: **[i, a] = S_j.**

2. **If GOTO (I_k, A) = I_j**, set the action as **state entry**: **[i, A] = j.**

3. **If I_i contains A → α·** (reduced production), set the action as **reduce**: [i, all entries] = R_p. Where **P** is the **production number** and **A → α** is the production rule.
4. If the state contains **S′ → S** then set the action to **accept** at that point: [i, $] = accept.

★ If any state has **Shift-Reduce (SR)** or **Reduce-Reduce (RR)** conflicts, it's called an **Inadequate State** and it can't be used in parsing.

SLR(1) Parser:

- **SLR(1)** is a more advanced version of **LR(0)**, designed to handle some extra complexities.
- It handles **Shift-Reduce (SR)** and **Reduce-Reduce (RR)** conflicts by considering **FOLLOW sets**.

Conflicts in SLR(1) Parser:

1. Shift-Reduce (SR) Conflict:

❖ Happens when in a state, both **shift** and **reduce** actions are possible.
Example:

■ **A → α·xβ** (shift)

■ **B → γ·** (reduce)

❖ **Condition**: This conflict can be avoided if **FOLLOW(B)** and **{x}** do not intersect (i.e., the terminal set after the symbol doesn't clash with the **FOLLOW** set of a non-terminal).

2. Reduce-Reduce (RR) Conflict:

❖ Occurs when two **reduce** actions are possible in the same state.

➢ **Example**:

■ **A → α·** (reduce)

■ **B → γ·** (reduce)

❖ **Condition**: This conflict can be avoided if **FOLLOW(A)** and **FOLLOW(B)** do not overlap (i.e., no common terminal symbols in their FOLLOW sets).

SLR(1) Parsing Table Construction:

1. **SLR(1) Table** is constructed in a similar way to **LR(0)** except for the **reduced entries**.

2. If a state **I_i** contains a reduced production like A → α·, we find the **FOLLOW(A)** set. For every element in **FOLLOW(A)**, we set the action [i, a] = R_p where **P** is the production number.

Important Points on SLR(1):

1. **SLR(1) is more powerful** than **LR(0)** because it can handle a wider range of grammars.

2. The **size of the SLR(1) parsing table** is the same as the size of the **LR(0)** parsing table.

3. **Every LR(0) grammar is also an SLR(1) grammar**, but not every **SLR(1)** grammar is **LR(0)**.

Canonical LR(1) Parser or (CLR(1))

➢ CLR(1) parser is the **most powerful** among LR parsers (LR(0), SLR(1), LALR(1), CLR(1)).

➢ It uses **1-symbol lookahead** while constructing the parsing table.

- Each item is written as:
  $A \to \alpha \cdot \beta , a$
  where a is the **lookahead symbol**.
- CLR(1) can handle a larger class of grammars than LR(0) and SLR(1), but the size of the parsing table is **very large**.

CLR(1) Parsing Table Construction:

i) Table construction is the same **as SLR(1)** for shift and goto entries.
ii) If state I_i contains a reduce item like **$A \to a\cdot, \$|a|b$**, then reduce action R_p (where P is production number) is added at

- $[i, \$] = R_p$
- $[i, a] = R_p$
- $[i, b] = R_p$

This means **reduce A → a** is done only when **lookahead is \$, a, or b**.

Conflicts in CLR(1):

- SR (Shift-Reduce) conflict:
  Example:
  $A \to \alpha \cdot x\beta , a$ (shift)
  $B \to \gamma \cdot , x \mid y$ (reduce)
  Conflict occurs when x is both after dot and in lookahead.
- RR (Reduce-Reduce) conflict:
  Example:
  $A \to \alpha \cdot , a_1$
  $B \to \gamma \cdot , a_2$

★ **If $a_1 \cap a_2 \neq \emptyset$, conflict occurs.**

NOTE:

- CLR(1) is the **most powerful** parser.
- CLR(1) can handle all LR(0), SLR(1), and LALR(1) grammars.
- But it is **not commonly used** due to the large **table size**.
- CLR(1) forms the **basis for the LALR(1)** parser.

Lookahead LR(1) Parser or (LALR(1))

- LALR(1) is made by **merging states** in CLR(1) which have the **same core** but different lookaheads.
- This reduces the size of the parsing table while maintaining **almost the same power** as CLR(1).
- LALR(1) is the **most commonly used LR parser in practice**.

Construction of LALR(1):

- States with the same core in CLR(1) are **merged** into one in LALR(1).

OPP < LL(1) < LR(0) < SLR(1) ≤ LALR(1) ≤ CLR(1)

- The lookaheads of merged states are **combined**.
- After merging, if **reduce** actions are added with overlapping lookaheads, **RR conflict** may occur.

Conflicts in LALR(1):

- **No SR conflict** occurs after merging.
- But **RR conflict** can occur when two **reduce actions** are possible for the **same lookahead symbol**.

Example:

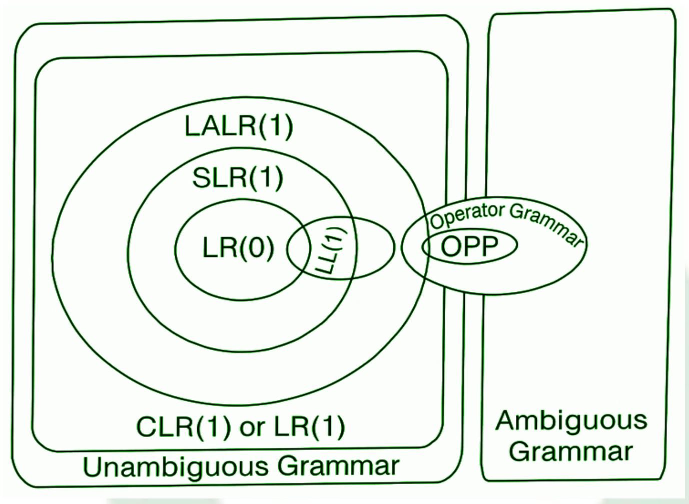
$A \to \alpha \cdot, a \mid t_1 \to$ reduce $A \to \alpha$
$B \to \gamma \cdot, a \mid t_2 \to$ reduce $B \to \gamma$

- If lookahead a is common in both **($t_1 \cap t_2 \neq \emptyset$)**, then **RR conflict** happens.

NOTE:

- **LALR(1) is less powerful than CLR(1)** but **more powerful than SLR(1)** and **LR(0)**.
- **Every LALR(1) grammar is also CLR(1)**. But CLR**(1) grammar is not LALR(1)**.
- **Parsing table size** is the same as SLR(1), which makes LALR(1) **efficient and practical**.

Parser Comparison :



- **OPP** = Operator Precedence Parser
- **LL(1)** = Top-down parser
- **LR(0), SLR(1), LALR(1), CLR(1)** = Bottom-up parsers

Set Relationship of Grammars:

➢ **LR(0) ⊆ SLR(1) ⊆ LALR(1) ⊆ CLR(1)**
➢ **LL(1) ⊆ CLR(1)**
➢ **Every LR(0) grammar is also a valid SLR(1), LALR(1), and CLR(1) grammar.**
➢ **Every LL(1) grammar is also CLR(1) grammar. But the reverse is not always true.**

NOTE:

★ **Number of states in SLR(1) = Number of states in LALR(1)**
★ **Number of states in CLR(1) ≥ LALR(1)**

★ **All these parsers handle unambiguous grammars only.**
★ **Ambiguous grammars cannot be parsed by LR-family parsers.**

4. Syntax Directed Translation

➢ SDT combines **Context-Free Grammar (CFG)** with **Semantic Actions (Transitions)**.

➢ The semantic actions are written on the **right-hand side** of production rules to specify the meaning of the syntax.

Applications of SDT:

1. **Semantic Analysis**: Checks the meaning of the program.
2. **Parse Tree Generation**: Creates a tree that shows how the program is structured.
3. **Intermediate Representation**: Converts the program into a format that is easier for a computer to process.
4. **Expression Evaluation**: Used to calculate or solve mathematical expressions.
5. **Infix to Prefix/Postfix Conversion**: Converts mathematical expressions from infix form to prefix or postfix form.

Example:

For the production rules:

- **S → S1 S2 [S.count = S1.count + S2.count]**
- **S → (S1) [S.count = S1.count + 1]**
- **S → ε [S.count = 0]**

Here, **count** is an **attribute** for the non-terminal **S** and stores information during the parsing process.

NOTE:

S → AB { print(*) }
A → a { print(1) }
B → S { print(2) }

★ Here, **print(*)**, **print(1)**, and **print(2)** are the semantic actions executed when the respective rules are applied.

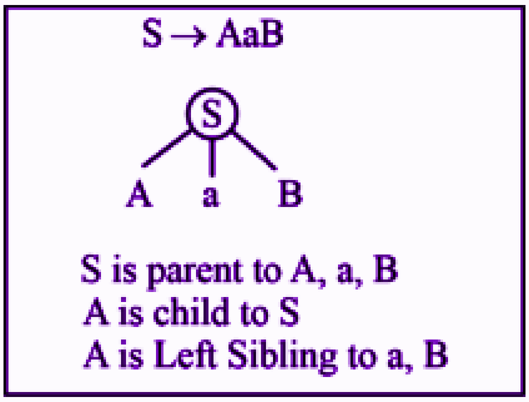
Steps to Construct SDT:

1. **Symbol Table**: Stores information about variables (names, types, etc.).
2. **Variable Declaration Checking**: Ensures all variables are properly declared.
3. **DAG Construction**: Builds a Directed Acyclic Graph (DAG) to optimize code.
4. **Type Checking and Conversion**: Ensures types are used correctly in expressions.
5. **Algebraic Expression Evaluation**: Calculates values for mathematical expressions.

★ An **annotated parse tree** shows the attributes for each node in the parse tree. This helps us understand the meaning of the program and the intermediate values at each step.

Attributes:

It is of two types.



1. **Inherited Attribute (RHS)**: The value of an inherited attribute is calculated based on the parent or sibling nodes in the syntax tree.
   - Example: **S → AaB {A.x = f(B.x | S.x)}**
   - In this case, the computation at the node depends on its parent (S) or siblings (A and B).

2. **Synthesized Attribute (LHS)**: The value of a synthesized attribute is calculated based on its children nodes in the syntax tree.
   - Example: **S → AB {S.x = f(A.x | B.x)}**
   - The computation depends on the children (A and B).

Syntax Directed Definition (SDD) :

It is of two types.

1. S-attributed SDD :

- Uses **only synthesized attributes**.
- The semantic actions (or computations) are placed at the **end** of the production rules.
- **Evaluation follows a bottom-up approach**.

Example:
For the rule:
A → BC { A.i = f(B.i or C.i) }
Here, A.i depends on B.i or C.i (children of the node).

2. L-attributed SDD

- Attributes are **synthesized** or **restricted to inherit** from **parent** or **left siblings**.
- **Evaluation follows a top-down approach**, typically following **Depth First Search (DFS)**.

Example:
For the rule:
A → BCD { C.i = f(A.i or B.i) } or
D.i = f(A.i or B.i or C.i) or
B.i = f(A.i)
These values depend on the children (left siblings or parents).

Important Points

- ❖ Every S-attributed SDT is also an L-attributed SDT.
- ❖ Not every L-attributed SDT is S-attributed.
- ❖ SDT follows a depth-first search (DFS) approach for evaluation.
- ❖ For **L-attributed evaluation**, use the **in-order traversal** of the annotated parser tree.
- ❖ For **S-attributed evaluation**, use the **reverse of the RMD (Rightmost Derivation) order.**

5. Intermediate Code Generation

Introduction:

- ➢ It is the **3rd phase** in the compiler (after syntax and semantic analysis).
- ➢ This phase **converts source code into an intermediate representation (IR)**.
- ➢ Intermediate code is **not machine code**, and **not high-level** either. It is something **in-between**.
- ➢ It is used to make the compiler **machine-independent**.
- ➢ If we have Intermediate Code, we can easily **generate code for multiple machines** from the same compiler.
- ➢ It helps in **optimization** (because this code is easier to analyze).
- ➢ Also used in **error detection** and **debugging**.

Types of Intermediate Representations:

Intermediate code can be represented in two main forms:

1. Linear Form:
Instructions are written one after another (like assembly code)

A. **Postfix code**: Also called Reverse Polish Notation (RPN)

B. **Three-address code (TAC)**: Each instruction has at most 3 operands

C. **Static Single Assignment (SSA)**: Every variable is assigned exactly once, with unique names (e.g., a1, a2, a3)

2. Non-Linear Form:
These are structured representations like trees or graphs

A. **Syntax Tree**: Tree representing structure of expression or program

B. **Directed Acyclic Graph (DAG)**: Used to detect common subexpressions

C. **Control Flow Graph (CFG)**: Shows how control flows between basic blocks

Example:
Expression : **(y + z) * (y + z)** Represent this in all form

COMPILER DESIGN

GATE फर्रे

Postfix Notation:

yz+yz+*

→ Uses operator precedence, no parentheses required.

3AC (Three Address Code):

$t_1 = y + z$

$t_2 = t_1 * t_1$

→ At most **3 addresses** used: LHS, operand1, operand2.

SSA (Static Single Assignment):

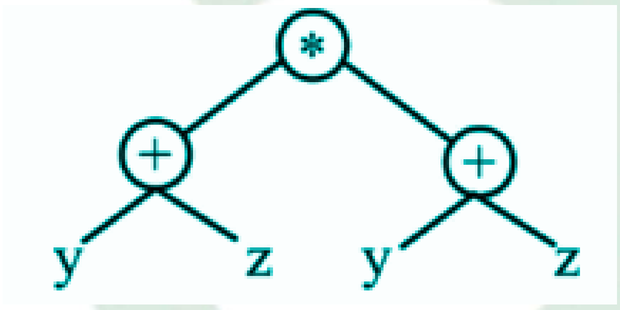
$t_1 = y + z$

$t_2 = t_1 * t_2$

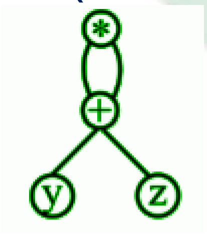
→ In SSA, once a variable is assigned **(t_1, t_2),** it cannot be reassigned again.

Syntax Tree:

Represents the full expression in a tree format based on operators and operands.



DAG (Directed Acyclic Graph):



→ Optimized version of syntax tree

→ Reuses **common subexpressions**, like **(y + z)**

→ Helps in **eliminating redundancy**

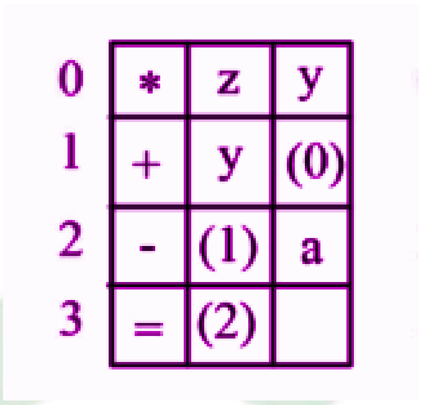
Three-address code (3-AC)

- Code where each statement has **at most 3 addresses**, including LHS
- Easy to generate and optimize
- Example: **$t_1 = a + b$, $t_2 = t_1 * c$**
- 3AC is always generated **using operator precedence**

Forms of 3AC Representation:

1. Triple Notation

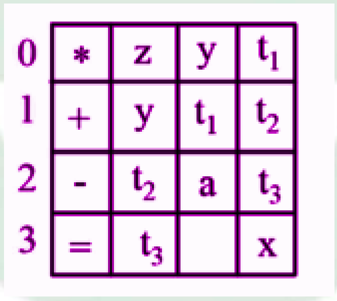
- Each row has: (Operator, Arg1, Arg2)
- Result is **position index**, not explicitly stored
- **Space efficient**, but **time inefficient** (hard to reorder instructions)



0	*	z	y
1	+	y	(0)
2	-	(1)	a
3	=	(2)	

2. Quadruple Notation

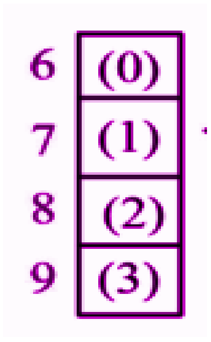
- Each row has: (Operator, Arg1, Arg2, Result)
- Result is **explicit**, easy to use and reorder
- **Time efficient**, but **uses more space**



0	*	z	y	t_1
1	+	y	t_1	t_2
2	-	t_2	a	t_3
3	=	t_3		x

3. Indirect Triple

- Maintains a **pointer table** to triples
- Easier to move instructions
- Improves flexibility in optimization



6	(0)
7	(1)
8	(2)
9	(3)

Static Single Assignment (SSA) Code:

- ➢ In SSA form, **each variable is assigned only once**.
- ➢ That means, whenever a variable is updated, we give it a **new version name (like x_1, x_2 or p, p_1, p_2).**
- ➢ **SSA** = Single meaning of each variable + 3-address code style

Example :

```
x = u - t
y = x * u
x = y + w
y = t - z
y = x * y
```

$\longrightarrow$

```
x = u - t
y = x * u
x₁ = y + w
y₁ = t - z
y₂ = x₁ * y₁
```

Find SSA ?

- **Already assigned:** u, t, w, z
- **New SSA vars:** x, y, x_1, y_1, y_2

  Total = 4 (already) + 5 (SSA) = 9 variables

Control Flow Graph (CFG)

- CFG shows the **flow of control** between different parts of a program.
- It is made of **nodes (basic blocks)** and **edges (jumps/branches)**.
- Each **basic block** is a group of statements where:
  - Control always enters from the first instruction (called the **leader**)
  - Control leaves only at the end — no jumps inside the block

Basic Block:

A **basic block** is a group of **3-address code** instructions with:

- No jump except at the end
- No label (target of jump) except at the beginning

```
1.   i = 1
2.   j = 1
3.   t1 = 5 * i
4.   t2 = t1 + 5
5.   t3 = 4*t2
6.   t4 = t3
7.   a[t4] = 1
8.   j = j + 1
9.   if j ≤ 5 goto 3
10.  i = i + 1
11.  if i < 5 goto 2
```

Leaders (start of each basic block):

1. First instruction **(i = 1)**
2. Any target of jump **(goto 3 → line 3, goto 2 → line 2)**
3. Instruction after a jump **(line 10 after line 9)**

Identified Basic Blocks:

- **LB1 → Line 1**
- **LB2 → Line 2**
- **LB3 → Lines 3 to 7**
- **LB4 → Line 8 and 9**
- **LB5 → Line 10**
- **LB6 → Line 11**

Control Flow (Edges):

LB1 → LB2
LB2 → LB3
LB3 → LB4
LB4 → LB3 (if j ≤ 5)
LB4 → LB5 (else)
LB5 → LB6
LB6 → LB2 (if i < 5)

This CFG shows **two loops:**

★ **Inner loop** on **j** → LB3 to LB4 back to LB3
★ **Outer loop** on **i** → from LB6 to LB2

6. Code Optimization

Introduction :

- The main goal is to save **time or memory** by improving the code.
- Optimization is based on **Flow Analysis**:
  - **Control Flow Graph (CFG)** – shows possible execution paths.
  - **Data Flow Graph (DFG)** – shows how data moves across the code.

Types of Optimization:

1. **Local Optimization**
   - Happens **inside a basic block** (a straight-line code with no jumps).
   - Called **Intra-procedural** optimization.
2. **Global Optimization**
   - Happens across **multiple blocks/functions** in the entire program.
   - Called **Inter-procedural** optimization.

Optimization Techniques:

1. Constant Folding
2. Copy Propagation
3. Strength Reduction
4. Dead Code Elimination
5. Common Sub-expression Elimination
6. Loop Optimization
7. Peephole Optimization

Constant Folding

- Evaluate constant expressions at compile time.

Example:
x = 2 * 3 + y
→ x = 6 + y (Folding done)
But if a variable is involved, like **x = 2 + y * 3,** folding cannot happen.

Copy Propagation

Replace variables with their assigned values.
i) Variable Propagation:
x = y
z = y + 2
z = x + 2 ← Replace y with x
ii) Constant Propagation:
x = 3
z = 3 + a
→ z = x + a

Strength Reduction

Replace expensive operations with cheaper ones.
Examples:

- **x = 2 * y → x = y + y (Addition is cheaper)**
- **x = 2 ^ y → x = y << 1 (Shift is much cheaper than exponentiation)**
- **x = y / 8 → x = y >> 3 (Right shift is cheaper than division)**

Dead Code Elimination

Remove code that will never execute.

Example:

x = 2
if (x > 2)
printf("code"); ← This will never run (dead code)
else
printf("optimization");

Since x > 2 is false, the printf("code") line will never run. It can be safely removed by the compiler.

Common Subexpression Elimination

If a computation is repeated, calculate it once and reuse.

Example:

x = (a + b) + (a + b) + c
→ t1 = a + b
→ x = t1 + t1 + c

This saves time and reduces redundant operations.

NOTE: DAG (Directed Acyclic Graph) is often used to identify and eliminate common subexpressions.

Loop Optimization

Used to improve performance by reducing the number of operations inside loops.

(i) Code Motion (Frequency Reduction)

Move statements that do not change within the loop, outside the loop.

```
for (i = 0; i < n; i++) {
    x = 10;        // Invariant
    y = y + i;
}
```

→

```
x = 10;            // Moved outside
for (i = 0; i < n; i++) {
    y = y + i;
}
```

(ii) Induction Variable Elimination

Remove unnecessary induction variables.

Example:

```
i1 = 0; i2 = 0;
for (i = 0; i < n; i++) {
    A[i1++] = B[i2++];
}
```

→

```
for (i = 0; i < n; i++) {
    A[i] = B[i];
}
```

Only one induction variable i is enough.

(iii) Loop Merging / Loop Jamming

Combine multiple loops with the same range to reduce overhead.

Example:

```
for(i = 0; i < n; i++) {
    A[i] = i + 1;
}
for(j = 0; j < n; j++) {
    B[j] = j - 1;
}
```

→

```
for(i = 0; i < n; i++) {
    A[i] = i + 1;
    B[i] = i - 1;
}
```

This saves execution time.

(iv) Loop Unrolling

Reduce loop control overhead by manually expanding the loop body.

Example:

```
for(i = 0; i < 3; i++) {
    printf("CD");
}
```

→

```
printf("CD");
printf("CD");
printf("CD");
```

Reduces loop variable comparison and increment operations.

Peephole Optimization

Looks at small sets of instructions (like through a "peephole") and replaces them with faster or shorter alternatives.

- ➔ Applied to intermediate or target code.
- ➔ Typical optimizations include:
  - ◆ Redundant instruction elimination
  - ◆ Strength reduction
  - ◆ Algebraic simplification
  - ◆ Use of registers
  - ◆ Jump optimization

7. Runtime Environments

➢ The runtime environment manages memory and control during program execution. It includes the structure of memory, registers, and control flow.

Types of Runtime Environments

1. **Fully Static Runtime Environment**
    - Suitable for languages without recursion or dynamic memory.
    - Activation record is fixed before execution.
    - Variables use fixed addresses.
    - Minimal bookkeeping.
    - No support for recursive calls.
2. **Stack-Based Runtime Environment**
    - Activation records are pushed when a function is called and popped after return.
    - Stack grows/shrinks with function calls.
    - Common in languages like C, Pascal.
3. **Fully Dynamic Runtime Environment**
    - Used in functional languages (Lisp, ML).
    - Activation records are created and destroyed dynamically.
    - Managed by garbage collector.
    - Uses heap-based memory.

Activation Records

- A block of memory used during the execution of a procedure.
- Created when a procedure is called and destroyed when it returns.

Contents of Activation Record:

1. Return Value
2. Static Link
3. Dynamic Link
4. Stack Pointer (SP) – Points to top of the stack
5. Return Address
6. Parameters (static and dynamic)
7. Local Variables
8. Local Arrays and Array Parameters
9. Intermediate Results
10. Saved Registers
11. Frame Pointer (FP) – Points to base of the current activation record

NOTE:
Activation records can be allocated in:

- Static area (e.g., Fortran 77)
- Stack area (e.g., C, Pascal)
- Heap area (e.g., Lisp)

STAR MENTOR CS/DA

KHALEEL SIR
ALGORITHM & OS
29 YEARS OF TEACHING EXPERIENCE

CHANDAN SIR
DIGITAL LOGIC
GATE AIR 23 & 26 / EX-ISRO

SATISH SIR
DISCRETE MATHEMATICS
BE in IT from MUMBAI UNIVERSITY

MALLESHAM SIR
M.TECH FROM IIT BOMBAY
AIR – 114, 119, 210 in GATE
(CRACKED GATE 8 TIMES)
14+ YEARS EXPERIENCE

VIJAY SIR
DBMS & COA
M. TECH FROM NIT
14+ YEARS EXPERIENCE

PARTH SIR
DA
IIIT BANGALORE ALUMNUS
FORMER ASSISTANT PROFESSOR

SAKSHI MA'AM
ENGINEERING MATHEMATICS
IIT ROORKEE ALUMNUS

SHAILENDER SIR
C PROGRAMMING & DATA STRUCTURE
M.TECH in Computer Science
15+ YEARS EXPERIENCE

AVINASH SIR
APTITUDE
10+ YEARS OF TEACHING EXPERIENCE

AJAY SIR
PH.D. IN COMPUTER SCIENCE
12+ YEARS EXPERIENCE