Last Minute Notes - Compiler Design

Last Updated : 25 Jan, 2025

In computer science, compiler design is the study of how to build a compiler, which is a program that translates high-level programming languages (like Python, C++, or Java) into machine code that a computer's hardware can execute directly. The focus is on how the translation happens, ensuring correctness and making the code efficient.

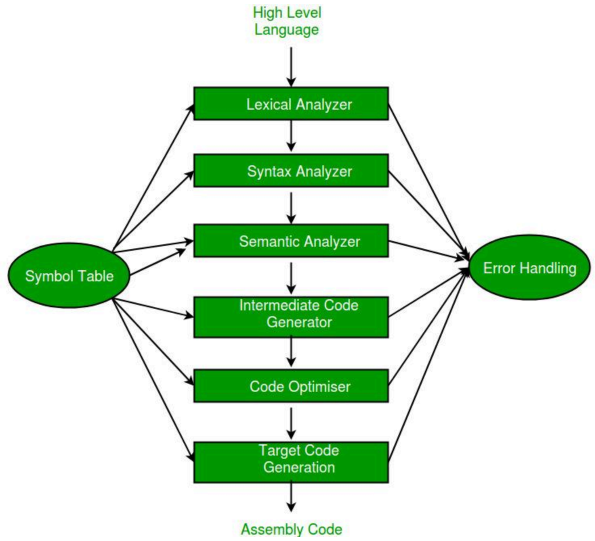
Compiler design is a core subject in computer science and plays a vital role in understanding how programming languages work at a deeper level.

Table of Content

Phases of a Compiler:

- **Lexical Analysis:** Tokenization of source code into meaningful units (tokens).
- **Syntax Analysis:** Construction of a parse tree based on grammar rules.
- **Semantic Analysis:** Ensures correctness of meaning (e.g., type checking).
- **Intermediate Code Generation:** Produces an intermediate representation (IR) for optimization and portability.
- **Code Optimization:** Enhances the efficiency of the intermediate code.
- **Code Generation:** Translates optimized IR into target machine code.

Read more about Phases of Compiler , Here.



Linking and Loading:

- **Linking:** The process of combining multiple object files and resolving symbolic references (such as function calls and variable accesses) to generate a single executable file.
- **Loading:** The process of placing the executable file into memory, resolving runtime addresses, and preparing it for execution by the CPU.

Read more about Difference Between Linker and Loader, [Here.]()

Lexical Analysis

Lexical analysis is the first phase of a compiler. It breaks the source code into small meaningful units called tokens.

Key Functions:

- **Tokenization:** Converts the source code into tokens (e.g., keywords, identifiers, operators, literals). Example: `int a = 5;` → Tokens: `int`, `a`, `=`, `5`, `;`
- **Removing Whitespaces and Comments:** These are ignored during token generation.
- **Error Detection:** Identifies errors like invalid symbols or unknown characters in the source code.

Components:

- **Lexical Analyzer (Lexer):** Performs the actual tokenization.
- **Symbol Table:** Stores information about variables, functions, and other identifiers.

Output of Lexical Analysis: A sequence of tokens is sent to the next phase (Syntax Analysis).

Token Categories in Lexical Analysis

Keywords:

- Reserved words with specific meaning in the language.
- Example: `int`, `if`, `while`, `return`.

Identifiers:

- Names given to variables, functions, arrays, etc.
- Example: `x`, `count`, `_value`.

Literals (Constants):

- Fixed values in the code.
- Example: `10`, `3.14`, `'a'`, `"hello"`.

Operators:

- Symbols used to perform operations.
- Example: `+`, `-`, `*`, `==`, `&&`.

Punctuation (Delimiters):

- Symbols that structure the program.
- Example: `;`, `,`, `()`, `{}`.

Special Symbols:

- Special-purpose symbols in some languages.
- Example: `#`, `$`.

Read more about Introduction of Lexical Analysis , Here.

Syntax Analysis and Parsing

Syntax analysis is the second phase of a compiler. It checks whether the tokens generated by lexical analysis follow the rules of the programming language's grammar.

Key Functions:

- **Parse Tree Construction:** Converts tokens into a hierarchical structure (parse tree) that represents the program's syntactic structure.
- **Grammar Validation:** Ensures the code adheres to the grammar rules of the language (e.g., correct placement of operators, brackets).
- **Error Detection:** Identifies syntax errors like missing semicolons or unmatched parentheses.
- **Input:** Sequence of tokens from the lexical analyzer.
- **Output:** Parse tree or syntax errors.

Types of Grammar Used:

- **Context-Free Grammar (CFG):** Used to define the syntax rules of programming languages.
- **Production Rules:** Defines how tokens can be combined (e.g., `E → E + T | T`).

Read more about Context Free Grammar, [Here].

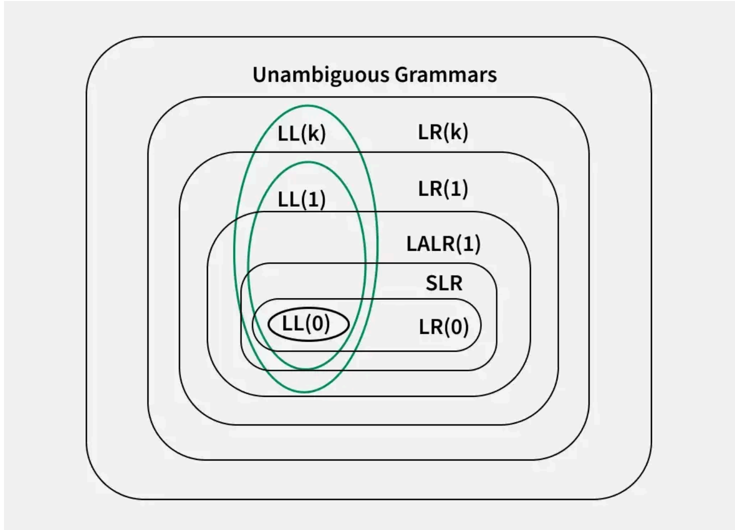
Classification of CFG:

- **Ambiguous Grammar:** A grammar is ambiguous if a string can have more than one derivation tree.
- **Unambiguous Grammar:** A grammar is unambiguous if every string has exactly one derivation tree.

Syntax Tree and Parse Tree

- **Parse Tree:** Represents the derivation of a string based on grammar rules. It contains all non-terminals and terminals.
  Read more about Parse Tree, [Here].
- **Syntax Tree:** Represents the semantic structure of the code. It focuses on essential elements (no redundant non-terminals).

Parser



A parser is a component of the compiler that performs syntax analysis.It checks whether the input tokens form a valid structure according to the grammar of the language. Output: A parse tree or syntax errors.

Classification of Parsers:

There are two types of parsers in compiler:

1. Top-Down Parsers: Build the parse tree from the root to the leaves.

Common Types:

- **Recursive Descent Parser**: Uses recursive functions for parsing.
- **LL Parser (Left-to-right, Leftmost derivation)**: Parses input from left to right, constructing the leftmost derivation. Example: LL(1) parser (1 lookahead token).

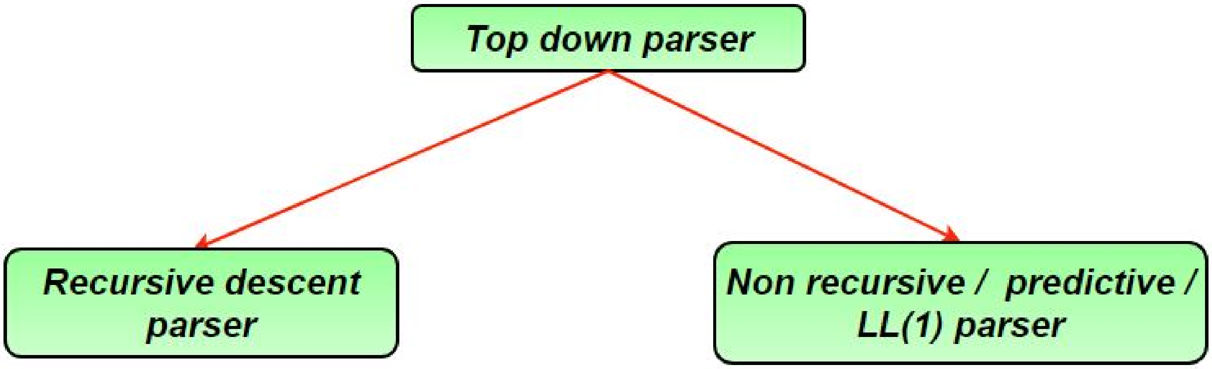
2. Bottom-Up Parsers: Build the parse tree from the leaves to the root.

Common Types:

- **Operator Precedence Parser:** A type of bottom-up parser that uses precedence and associativity rules of operators to decide shifts and reductions, suitable for parsing expressions in operator precedence grammars.
- **LR Parser (Left-to-right, Rightmost derivation)**: Parses input from left to right, constructing the rightmost derivation. Example: LR(0), SLR, CLR, LALR parsers.

Read more about Types of Parsers, [Here](#).

Top-Down Parser



A Top-Down Parser constructs the parse tree from root to leaves using a Leftmost Derivation (LMD). It predicts the next production to apply based on the input tokens.

LL(1) Parser

An LL(1) parser is a top-down parser that reads input Left-to-right, constructs a Leftmost derivation, and uses 1 lookahead token to decide parsing actions.

LL(1) Grammar: A grammar is said to be LL(1) if it can be parsed by a Top-Down Parser using Left-to-right scanning of input, producing a Leftmost Derivation, and requires only 1 lookahead symbol to decide which production to use at each step.

A grammar to be LL(1) must satisfy the following conditions:

1. For every pair of productions $A \rightarrow \alpha \mid \beta$, $First(\alpha) \cap First(\beta) = \emptyset$, i.e., $First(\alpha)$ and $First(\beta)$ should be two disjoint sets for every pair of productions.
2. If $First(\beta)$ contains ϵ and $First(\alpha)$ also contains ϵ, then $Follow(A) \cap First(\alpha) = \emptyset$

Steps to Construct LL(1) Parsing Table:

1. Remove Left Recursion: Rewrite rules to eliminate left recursion.

2. Left Factoring: Remove common prefixes in grammar rules.

3. Find First and Follow Sets:

- **First Set:** First terminal symbol derivable from a non-terminal.
- **Follow Set:** Terminals that can appear immediately after a non-terminal in derivations.

4. Construct Parsing Table: Use the First and Follow sets to fill the table.

Read more about Construction of LL(1) Parsing Table, [Here].

First and Follow Sets Calculation

1. First Set: The First Set of a variable contains the terminals that can appear as the first symbol in the strings derived from that variable.

Rules to Calculate First Set:

- If `X` is a terminal, `First(X) = {X}`.
- If `X → ε`, include `ε` in `First(X)`.
- If `X → Y1 Y2...Yn`, then:
- Add `First(Y1)` to `First(X)`, excluding `ε`.
- If `Y1` derives `ε`, check `Y2`, and so on.

2. Follow Set: The Follow Set of a variable contains terminals that can appear immediately after it in the input string.

Rules to Calculate Follow Set:

- Start symbol always has `$` in its Follow set.
- For a production `A → αBβ`:
    - Add `First(β)` (excluding `ε`) to `Follow(B)`.
- If `β → ε`, or `A → αB`, then:
    - Add `Follow(A)` to `Follow(B)`.

Read more about First and Follow in Compiler Design, Here.

Example: Consider the Grammar:

E --> TE'
E' --> +TE' | ε
T --> FT'
*T' --> *FT' | ε*
F --> id | (E)

**ε denotes epsilon*

Step 1: The grammar satisfies all properties in step 1.

Step 2: Calculate first() and follow().

Find their First and Follow sets:

	First	Follow
E –> TE'	{ id, (}	{ $,) }
E' –> +TE'/ ε	{ +, ε }	{ $,) }
T –> FT'	{ id, (}	{ +, $,) }
T' –> *FT'/ ε	{ *, ε }	{ +, $,) }
F –> id/(E)	{ id, (}	{ *, +, $,) }

Step 3: Make a parser table.

Now, the LL(1) Parsing Table is:

	id	+	*	(	)	$
E	E –> TE'			E –> TE'		
E'		E' –> +TE'			E' –> ε	E' –> ε
T	T –> FT'			T –> FT'		
T'		T' –> ε	T' –> *FT'		T' –> ε	T' –> ε
F	F –> id			F –> (E)		

Recursive Descent Parser

A Recursive Descent Parser is a type of Top-Down Parser that uses recursive functions to process the input and construct the parse tree.

Key Features:

1. **Parsing Direction:** Left-to-right on the input.
2. **Derivation:** Constructs Leftmost Derivation.
3. **Implementation:** Uses a set of mutually recursive functions, one for each non-terminal in the grammar.

Steps in Recursive Descent Parsing:

1. Start with the start symbol of the grammar.
2. For each non-terminal, call a corresponding recursive function.
3. For each terminal, match it with the input token.
4. Backtrack if there's a mismatch (limited capability without modifications).

Read more about Recursive Descent Parser, Here.

Bottom-Up Parser

Operator Precedence Parser:

An operator precedence parser is a bottom-up parser that interprets an operator grammar. This parser is only used for operator grammars.A grammar is said to be operator precedence grammar if it has two properties:

- No R.H.S. of any production has a ∈.
- No two non-terminals are adjacent.

Operator Precedence Relation:

a ⋗ b means that terminal "a" has the higher precedence than terminal "b".
a ⋖ b means that terminal "a" has the lower precedence than terminal "b".
a ≐ b means that the terminal "a" and "b" both have same precedence.

Read more about Operator Precedence Grammar and Parser, Here.

The operator precedence table for the grammar will be-

	+	x	id	$
+	⋗	⋖	⋖	⋗

x	⋗	⋗	⋖	⋗
id	⋗	⋗	—	⋗
$	⋖	⋖	⋖	A

Operator Precedence Parser Algorithm :

1. If the front of input $ and top of stack both have $, it's done
else
2. compare front of input b with ⋗
if b! = '⋗'
then push b
scan the next input symbol
3. if b == '⋗'
then pop till ⋖ and store it in a string S
pop ⋖ also
reduce the popped string
if (top of stack) ⋖ (front of input)
then push ⋖ S
if (top of stack) ⋗ (front of input)
then push S and goto 3

In Bottom-Up Parsing, the following types of entries/actions are used to guide parsing:

1. **Shift:** Move the next input symbol onto the stack.
2. **Reduce:** Replace a sequence of symbols on the stack (matching the right-hand side of a production) with the corresponding non-terminal (left-hand side).
3. **Accept:** Indicates successful parsing when the start symbol is reduced and the input is fully consumed.

LR Parser

An LR Parser is a Bottom-Up Parser that reads the input Left-to-right and constructs a Rightmost Derivation in reverse.

1. LR(0) Parser : Closure() and goto() functions are used to create canonical collection of LR items. Conflicts in LR(0) parser :

1. **Shift Reduce (SR) conflict :** When the same state in DFA contains both shift and reduce items. A -> B . xC (shifting) B -> a. (reduced)
2. **Reduced Reduced (RR) conflict :** Two reductions in same state of DFA A -> a. (reduced) B -> b. (reduced)

2. SLR Parser : It is powerful than LR(0). Ever LR(0) is SLR but every SLR need not be LR(0). Conflicts in SLR

- **SR conflict :** A -> B . xC (shifting) B -> a. (reduced) if FOLLOW(B) ∩ {x} ≠ φ
- **RR conflict :** A -> a. (reduced) B -> b. (reduced) if FOLLOW(A) ∩ FOLLOW(B) ≠ φ

3. CLR Parser : It is same as SLR parser except that the reduced entries in CLR parsing table go only in the FOLLOW of the l.h.s non-terminal.

4. LALR Parser : It is constructed from CLR parser, if two states having same productions but may contain different look-aheads, those two states of CLR are combined into single state in LALR. Every LALR grammar is CLR but every CLR grammar need not be LALR.

Steps for LR Parsing Table Construction:

1. Augment the Grammar: Add a new production `S' → S`, where `S` is the start symbol.

2. Construct Canonical LR(0) Items: Create **item sets** (closures and GOTO operations).

3. Compute Parsing Table:

- **Action Table:** Contains shift, reduce, accept, or error.
- **Goto Table:** Specifies transitions for non-terminals.

4. Conflict Checking: Ensure no shift/reduce or reduce/reduce conflicts.

Parsers Comparison : LR(0) ⊂ SLR ⊂ LALR ⊂ CLR LL(1) ⊂ LALR ⊂ CLR If number of states LR(0) = n1, number of states SLR = n2, number of states LALR = n3, number of states CLR = n4 then, n1 = n2 = n3 <= n4 .

Read more about LR Parser, Here.

Syntax Directed Translation

Syntax Directed Translation (SDT) combines Context-Free Grammar (CFG) with semantic rules to assign meaning or perform actions during parsing.

Attributes in SDT

- **Inherited Attributes:**
    - Depend on parent or sibling nodes.
    - Example: `x` is inherited in `A → B {A.x = B.x + 2}`.
- **Synthesized Attributes:**
    - Depend on child nodes.
    - Example: `x` is synthesized in `A → B {A.x = B.x + 2}`.

Syntax Directed Definitions (SDD)

L-Attributed Grammar: Attributes are either: Synthesized OR Restricted Inherited (from parent or left siblings only).

- **Evaluation Order:** Topological (In-Order traversal). Example: `S → AB {A.x = S.x; B.x = f(A.x)}`.

S-Attributed Grammar: Only Synthesized Attributes are used.

- **Evaluation Order:** Reverse Rightmost Derivation (Bottom-Up). Example: `E → E1 + T {E.val = E1.val + T.val}`.

Read more about S-Attributed and L-Attributed in SDTs, [Here].

Attribute Examples:

1. Inherited Attributes Example:

```
D → T L {L.in = T.type}
T → int {T.type = int}
L → id {AddType(id.entry, L.in)}
```

`L.in` is inherited, and `T.type` is synthesized.

2. Synthesized Attributes Example:

```
E → E1 + T {E.val = E1.val + T.val}
T → int {T.val = int}
```

`E.val` and `T.val` are synthesized.

- **Synthesized** → Bottom-Up Evaluation.
- **L-Attributed** → Includes Synthesized + Restricted Inherited evaluated In-Order.

Intermediate Code Generation and Optimization

Three-Address Code (3AC):

- Code representation where each statement has at most 3 operands, including the LHS.
- **Applications of 3AC:**
  1. Operator precedence parsing is used.
  2. Intermediate code representation.
  3. Example:

     ```
     u = t - z
     v = u * w
     w = v + t
     ```

     Minimum variables required: Optimize the number of temporary variables for efficiency.

Read more about 3AC, [Here]().

Static Single Assignment (SSA) Code:

- **Definition:** Every variable in the code has a **single assignment**.
- **Characteristics:**
  - Simplifies optimization.
  - Uses new names (e.g., `x`, `p1`, `q1`) for reassignments.
  - Example:

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

  - Variables `[u, t, v, w, z]` are already assigned, so we can't reuse them.

  Equivalent SSA Code:

```
x = u - t
y = x * v
p = y + w
q = t - x
r = p * q
```

Total Variables: 10.

Read more about SSA, Here.

Control Flow Graph (CFG):

- **Definition:** CFG represents a program as nodes (basic blocks) and edges (control flow).
- **Basic Block:** A sequence of instructions with:
  1. One entry point (leader).
  2. One exit point.
- **Application:** Identifies and optimizes independent code blocks.

Code Optimization:

Objective: Reduce execution time and memory usage.

Techniques:

1. Constant Folding: Evaluate constant expressions at compile time. Example: `x = 2 * 3 + y → x = 6 + y`.

2. Copy Propagation: Replace redundant variables. Example: `z = y + 2 → z = x + 2` (if `x = y`).

3. Strength Reduction: Replace expensive operations with cheaper ones. Example: `x = 2 * y → x = y + y`.

4. Dead Code Elimination: Remove code that does not affect the output. Example: Remove `if (false) { ... }`.

5. Common Subexpression Elimination: Eliminate repeated calculations using DAGs. Example:

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

6. Loop Optimization:

- **Code Motion:** Move invariant code outside loops.
- **Induction Variable Elimination:** Replace variables with simpler expressions.
- **Loop Jamming:** Combine multiple loops.
- **Loop Unrolling:** Reduce loop overhead by executing multiple iterations in a single iteration.

7. Peephole Optimization:

Analyze short sequences of code (peepholes) and replace them with faster alternatives. Applied to intermediate or target code.
Following Optimizations can be used:

- *Redundant instruction elimination*
- *Flow-of-control optimizations*
- *Algebraic simplifications*
- *Use of machine idioms*

Read more about Code Optimization in Compiler Design, Here.

Programmers, write computer programs that make certain tasks easier for users. This program code is written in High-Level Programming languages like C, C++, etc....

6 min read

Difference Between Native Compiler and Cross Compiler

Compilers are essential tools in software development, helping to convert high-level programming languages into machine-readable code. Among various types of...

5 min read

Code Optimization in Compiler Design

Code optimization is a crucial phase in compiler design aimed at enhancing the performance and efficiency of the executable code. By improving the quality of the...

9 min read

Error Handling in Compiler Design

During the process of language translation, the compiler can encounter errors. While the compiler might not always know the exact cause of the error, it can detect and...

5 min read

Labeling Algorithm in Compiler Design

Labeling algorithm is used by compiler during code generation phase. Basically, this algorithm is used to find out how many registers will be required by a program to...

3 min read

Basic Blocks in Compiler Design

Basic Block is a straight line code sequence that has no branches in and out branches except to the entry and at the end respectively. Basic Block is a set of statements that...

3 min read

BNF Notation in Compiler Design

BNF stands for Backus Naur Form notation. It is a formal method for describing the syntax of programming language which is understood as Backus Naur Formas...

3 min read

Parse Tree in Compiler Design

In compiler design, the Parse Tree depicts the syntactic structure of a string in accordance with a given grammar. It was created during the parsing phase of...

4 min read

Machine Independent Code optimization in Compiler Design

Machine Independent code optimization tries to make the intermediate code more efficient by transforming a section of code that doesn't involve hardware components...

7 min read