BAHIR DAR UNIVERISTY

Faculty of Computing: Department of Software Engineering

Principles of Compiler Design

Name= Solome Andarge
ID= BDU1507772
Section= B

Submitted to: Mr. Wondimu.
Submission date: 27-04-2018 E.C

Assignment 01 – Compiler Design / Syntax Analysis

Student 71:-

1. (Theory): Explain the difference between top-down and bottom-up parsers.
2. (C++): Write a C++ program to check whether a string contains balanced curly braces {}.
3. (Problem-solving): Grammar: $S \rightarrow aSb \mid \varepsilon$
   Draw the parse tree for "aaabbb".

1. Top-Down vs. Bottom-Up Parsing

Top-down parsing builds the parse tree by predicting productions from the start symbol, while bottom-up parsing reconstructs the derivation by reducing input tokens to nonterminals. Bottom-up parsers are more powerful and flexible, especially for complex grammars with left recursion.

Top-Down Parsing (LL Parsers)

Core Idea:
Starts from the **start symbol** and tries to match the input by expanding nonterminals using grammar rules.

How it works:

- Predicts which production to use based on the next input token (lookahead).
- Builds the parse tree **from root to leaves**.
- Uses recursive procedures or parsing tables.

Key techniques:

- **Recursive Descent:** Manual implementation using functions for each nonterminal.
- **LL(1) Parsing Table:** Uses a single lookahead token to decide which rule to apply.

Grammar constraints:

- Must be **left-factored** (no ambiguity in choosing rules).
- Cannot have **left recursion** (e.g., $A \rightarrow A\alpha \mid \beta$), which causes infinite loops.

Error handling: Detects errors **early**, often as soon as an unexpected token is encountered.

Use cases: Simple, hand-written parsers.

Bottom-Up Parsing (LR Parsers)

Core Idea:
Starts from **input tokens** and tries to reduce them to the start symbol by applying grammar rules in reverse.

How it works:

- Uses a **stack** to hold symbols and a **parsing table** to decide actions.
- Actions include **shift** (push token) and **reduce** (replace RHS of rule with LHS).
- Builds the parse tree **from leaves to root**.

Key techniques:

- **Shift-Reduce Parser:** Most common bottom-up strategy.
- **SLR(1), LALR(1), Canonical LR:** Variants with increasing power and complexity.

Grammar flexibility:

- Can handle **left-recursive** and **ambiguous grammars**.

- Suitable for **almost all deterministic context-free grammars (works** with a wider range of grammars).

Error handling:

- Errors may be detected **later**, after several shifts and reductions.

Use cases:

- Industrial-strength compilers (e.g., GCC).
- Automatically generated parsers using tools like YACC, Bison.

Comparison Table:

Feature	**Top-Down (LL)**	**Bottom-Up (LR)**
Start point	Start symbol	Input tokens
Parse direction	Root → Leaves	Leaves → Root
Grammar type	LL(1)	LR, SLR, LALR

Handles left recursion?	No	Yes
Error detection	Early (predictive)	Later (after reductions)
Example parser	Recursive Descent	Shift-Reduce

2. C++ Program: Check Balanced Curly Braces {}

```
#include <iostream>
#include <stack>
#include <string>

bool areBracesBalanced(const std::string& str) {
    std::stack<char> s;

    for (char ch : str) {
        if (ch == '{') {
            s.push(ch);
        } else if (ch == '}') {
            if (s.empty()) return false;
            s.pop();
        }
    }

    return s.empty();
}

int main() {
    std::string input;
    std::cout << "Enter a string with curly braces: ";
    std::getline(std::cin, input);

    if (areBracesBalanced(input)) {
        std::cout << "Curly braces are balanced.\n";
    } else {
        std::cout << "Curly braces are NOT balanced.\n";
    }

    return 0;
}
```

Sample Runs:

```
Input: {a{b}c}
Output: Curly braces are balanced.

Input: {a{b}c
Output: Curly braces are NOT balanced.
```

3. Problem-Solving: Parse Tree

Grammar: S → aSb | ε
Input string: aaabbb

Step 1: Derivation

S
→ aSb
→ aaSbb
→ aaaSbbb
→ ε

Step 2: Parse Tree

```
           S

         /  |   \

       a   S    b

          /|\

        a S b

           /|\

         a S b

            |

            ε
```

Explanation:

- Root is S.
- Each a branches left, each b branches right.
- The deepest S goes to ε.

- Leaves (terminals) are: a a a b b b.

  (a a a ε b b b → concatenates to aaabbb.

Assignment-02: Difference between One-Pass and Multi-Pass Compilers

Introduction

What is a Compiler?

A **compiler** is a system software (**translator program**) that translates **high-level programming languages** (like C, Java) into **machine code** or **intermediate code** that a computer can understand and execute.

➢ Understanding compiler passes helps us:

- Design efficient compilers
- Improve program performance
- Understand how modern programming languages work internally

➢ Why Compiler Passes Matter

- Compiler design affects **speed, memory usage, optimization, and error handling**
- One-pass and multi-pass compilers represent **different design strategies** which affects how the source code is analyzed and converted into machine code.

➢ **The compilation process** is typically divided into several **stages (phases)** each responsible for a specific task.

1. Lexical Analysis: - Converts characters into tokens (identifiers, keywords, and operators).
2. Syntax Analysis: - Checks the program's grammatical structure against language rules.
3. Semantic Analysis: - Checks meaning and ensures code makes sense (type checking, declarations, scope rules).
4. Intermediate Code Generation:- Produces machine-independent code
5. Code Optimization: - Improves performance and efficiency by reducing instructions, memory usage, and execution time.
6. Code Generation:- Produces final machine code

Key Question: Are these phases executed **once** or **multiple times**?
This is the central difference between **one-pass** and **multi-pass compilers**. There difference lies in how these phases are organized and executed.

The term **pass** refers to one complete traversal of the source code or its intermediate representation. Depending on how many times the compiler traverses the code; compilers are classified as **one-pass** or **multi-pass**. Understanding this distinction is crucial because it directly affects compilation speed, memory usage, optimization capabilities, and the complexity of language features supported.

What is a One-Pass Compiler?

Definition

A **one-pass compiler** processes the entire source code **only once** or in a **single traversal**, performing all compilation phases in a single pass and generates the target code during the same pass. As it reads the program line by line, it simultaneously performs lexical analysis, parsing, semantic checks, and generates machine code.

Key Characteristics

- Reads source code from start to end and performs all tasks **once** (does not revisit earlier parts of the program).
- Generates target code immediately and Performs analysis and code generation simultaneously (requires immediate decisions about symbol resolution and code generation).
- Cannot easily handle forward references (e.g., calling a function before it is defined).

Key Idea:

“Analyze and translate at the same time.”

Simple Flow: -

Source Code → Compiler (Lexical → Syntax → Semantic → Code Gen) → Target Code

➢ **Why Forward Declarations Are a Problem**

In one-pass compilers:

- Variables or functions must be declared **before use**
- The compiler cannot “remember” future code

Example Scenario (Where One-Pass Compilers Are Used)

- Early programming languages (**Pascal compilers**)
- Simple embedded systems
- Small educational compilers

Code Constraint Example

```
int main() {
    print(x); // ERROR: x not declared yet

    int x = 10;

}
```

- The compiler encounters `print(x)` first
- `x` is not yet declared
- One-pass compiler throws an error

One-pass compiler **cannot handle forward declarations easily**

Advantages & Disadvantages of One-Pass Compilers

Advantages

- **Fast compilation:** Since the source code is read only once, compilation is fast.
- **Low memory usage:** No need to store intermediate structures or representations.
- **Simple design:** Easy to implement and understand

Disadvantages

- **Poor optimization:** Cannot analyze the entire program globally.
- **Weak error handling:** Some errors detected late or incorrectly.
- **Not suitable for complex languages**

What is a Multi-Pass Compiler?

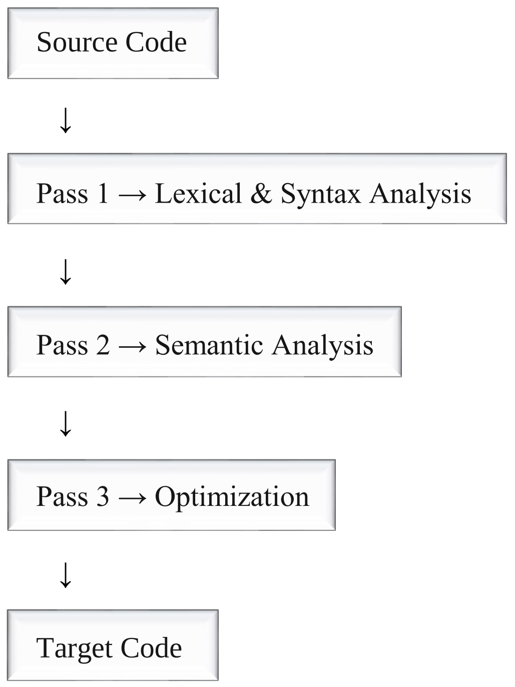
Definition

A **multi-pass compiler** processes the source code **multiple times**, where each pass focusing on a specific compilation phase performing specific task.

Key Characteristics

- Divides compilation into logical phases
- First pass analyzes structure
- Later passes refine meaning, optimize, and generate code
- Uses intermediate representations (IR) between passes
- Supports advanced optimizations and complex language features.

Flow Diagram


Source Code
Pass 1 → Lexical & Syntax Analysis
Pass 2 → Semantic Analysis
Pass 3 → Optimization
Target Code

Multi-Pass Compiler Example

Most modern compilers are multi-pass:

- **GCC (GNU Compiler Collection)**
- **LLVM**
- **Java Compiler (javac)**

Why Multi-Pass Works Better

- Can handle forward references
- Performs global optimizations
- Produces optimized, efficient machine code
- Easier to maintain and extend
- Allows deep analysis of entire program

The compiler collects all declarations first, and then checks usage later avoiding early errors.

Advantages & Disadvantages of Multi-Pass Compilers

Advantages

- Highly optimized code
- Supports complex language features (like OOP, templates)
- **Strong error detection:** Errors found accurately
- **Advanced optimization:** Faster, smaller executables
- **Modular design:** Easy to extend or modify

Disadvantages

- Slower compilation → multiple scans of source code.
- Higher memory usage → stores intermediate representations.
- More complex design or implementation

One-Pass vs Multi-Pass Compiler – Direct Comparison

Feature	**One-Pass Compiler**	**Multi-Pass Compiler**
Number of Passes	One	Multiple
Speed	Faster	Slower
Optimization	Minimal	Advanced
Error Handling	Limited	Strong
Complexity	Simple	Complex
Usage	Small systems, simple languages	Modern compilers, complex languages

This comparison helps understand **why modern compilers prefer multi-pass design**.

- One-pass: speed & simplicity
- Multi-pass: accuracy & performance

The choice depends on:

- Language complexity
- System resources
- Required optimization level

Importance in Modern Compiler Design

- Modern software requires:
    - Performance optimization
    - Scalability
    - Cross-platform support (Portability)
    - Advanced error diagnostics (debugging)

Hence, **Multi-pass compilers dominate modern compiler construction as they:**

- Support multiple architectures
- Enable aggressive optimizations
- Maintain clean, modular compiler design

Examples:

- Mobile apps
- Operating systems
- Game engines
- AI and scientific computing

Conclusion

Summary

- One-pass compilers are fast, simple, and have limited optimization, limited error handling.
- Multi-pass compilers are powerful, flexible, optimized and strong error handlers.
- Choice depends on **language complexity and system requirements**
- **Modern languages require multi-pass compilation** due to complexity and optimization needs.
- Compiler design directly affects program quality, performance, and maintainability

Understanding this helps developers write **better, more efficient code** and debug errors more effectively.