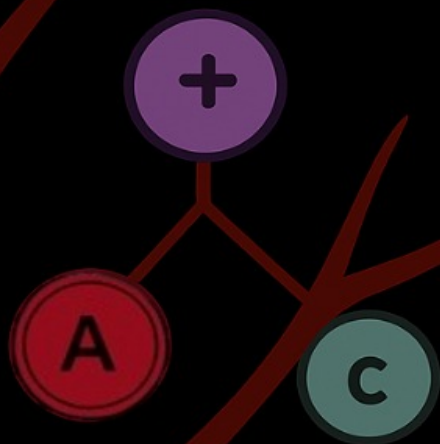# COMPILER BLUEPRINT

An Academic Guide to the Concepts of Compiler Design

Md. Naimur Rahman

Sajjad Hossen Shemanto

Compiler Blueprint

An Academic Guide to the Concepts of Compiler Design

Mastering Concepts and Coding: Your Path to Academic Excellence

by

Md. Naimur Rahman
&
Sajjad Hossen Shemanto

Copyright

Author and Publisher:
Md. Naimur Rahman and Sajjad Hossen Shemanto
Printed locally by the Author
Dhaka, Bangladesh

Cover Design, Content, and Layout by:
Md. Naimur Rahman and Sajjad Hossen Shemanto
Email: rahman23105101275@diu.edu.bd
Email: shemanto23105101501@diu.edu.bd

Includes Project: Compiler Learning Hub
🌐 compiler-learninghub.onrender.com

Available in both print and digital formats.

First Edition: August 2025

Dedication

To those
who never had the chance to speak,
and to those
whose voices were never heard —
this book is for you.

Acknowledgements

We are deeply thankful to the Almighty for giving us the strength and motivation to complete this book.

We express our heartfelt gratitude to our mentor and respected teacher, **Mushfiqur Rahman, Assistant Professor, Daffodil International University**, whose guidance and encouragement helped us throughout the process.

We also acknowledge our friends, classmates, and family members for their endless support and inspiration.

Preface

This book, *"Compiler Blueprint: An Academic Guide to the Concepts of Compiler Design"*, is intended for students and beginners who want to explore how compilers work in practice.

Compiler design is often considered complex.Our aim was to break it down into simple, step-by-step explanations—supported with clear examples, parse trees, grammars, and LaTeX-formatted diagrams.

Each chapter is designed to be beginner-friendly, and ends with problems and exercises to enhance understanding. We hope that this book will serve as both a learning tool and a reference guide for your compiler journey.

<div align="right">

Md Naimur Rahman
Sajjad Hossen Shemanto
August 2025

</div>

Contents

Chapter 1

Introduction to Compilers

1.1. Compiler

A **compiler** is a special type of software that translates source code written in a high-level programming language (such as C, C++ or Java) into machine-level code that the computer's processor can execute.

This process is essential because computers cannot understand high-level languages directly — they only understand binary instructions (0s and 1s).

The compiler performs this translation in several stages, including lexical analysis, syntax analysis, semantic analysis, optimization, and code generation.

Role of a Compiler in Programming

The compiler plays several key roles in the software development process:

- **Translation:** Converts high level language into machine code.

- **Error Checking:** Identifies syntax and semantic errors in the source code.

- **Optimization:** Improves the efficiency of the generated code.

- **Portability:** Enables the same code to be compiled on different hardware platforms.

- **Abstraction:** Allows programmers to write in human readable language without worrying about hardware details.

Without compilers, software development would be time-consuming and error-prone, as developers would have to write programs in machine code manually.

Compilation vs Execution

Compilation is the process where the source code is transformed into machine code, while **execution** is the actual running of that machine code on a computer.

These are two separate steps:

- **Compilation:** Happens once; creates an executable file.

- **Execution:** Happens every time the user runs the program.

Analogy: Compilation is like cooking a dish and putting it in a lunchbox. Execution is eating it. You don't need to cook again every time — just eat what's ready.

Real-World Analogy (Translator Example)

Imagine that a book written in Bengali needs to be read by an English speaker. A human translator translates the entire book into English and gives the translated version to the reader. Now, the reader can read the translated book any time without needing the translator again.

Similarly, a compiler reads the entire program, translates it into machine code (executable), and produces a version that the computer can run without needing to read the original source again.

This analogy helps explain the difference between a compiler and an inter-

preter — an interpreter would read and translate *line by line* every time, like a live translator.

1.2. Language Processors

1.2.1 Compiler

Simply stated, a compiler is a program that can read a program in one language–the source language–and translate it into an equivalent program in another language–the target language; see Fig. 1.1. An important role of the compiler is to report any errors in the source program that it detects during the translation process.

source program

Compiler
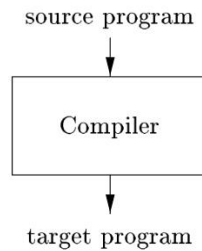
target program

Figure 1.1: A compiler

If the target program is an executable machine-language program, it can then be called by the user to process inputs and produce outputs; see Fig. 1.2.

input → Target Program → output
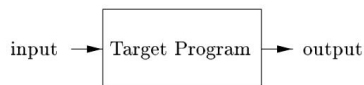
Figure 1.2: Running the target program

1.2.2   Interpreter

An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations specified in the source program on inputs supplied by the user, as shown in Fig. 1.3.

```
source program  →  ┌──────────────┐
                    │  Interpreter │  →  output
       input     →  └──────────────┘
```
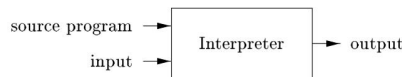
Figure 1.3: An interpreter

The machine-language target program produced by a compiler is usually much faster than an interpreter at mapping inputs to outputs . An interpreter, however, can usually give better error diagnostics than a compiler, because it executes the source program statement by statement.

1.2.3   Hybrid compiler

Java language processors combine compilation and interpretation, as shown in Fig. 1.4. A Java source program may first be compiled into an intermediate form called bytecode. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network. In order to achieve faster processing of inputs to outputs, some Java compilers, called just-in-time compilers, translate the bytecodes into machine language immediately before they run the intermediate program to process the input.

> **NOTE**
>
> Python is similar to Java in combining compilation and interpretation, but there are differences between the two languages
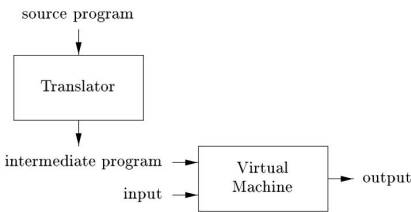
Figure 1.4: A hybrid compiler

1.2.4   Compiler vs Interpreter

Compilers and interpreters are both language translators, but they work differently. The following table highlights their key differences.

Feature	Compiler	Interpreter
Execution	Translates whole program at once	Translates one line at a time
Speed	Faster after compilation	Slower since it processes line-by-line
Output	Creates an executable file	No file created
Error Handling	Shows all errors after compiling	Stops at the first error
Examples	C, C++, Java	Python, JavaScript, Ruby

Table 1.1: Comparison Between Compiler and Interpreter

1.3. The Language Processing System

When a program written in a high-level language is executed, it goes through multiple stages and tools. These tools together form the **language processing system**.

Source Program

The source program is the initial code written by a programmer using a high-level language such as C, C++, or Java. It contains the logic, control flow, and data structures needed to perform specific tasks. However, computers cannot directly execute high-level instructions. Therefore, the source program must be translated into machine-level code. The source file typically has extensions like `.c`, `.cpp`, or `.java`. This program is passed to a series of language processing tools starting with the preprocessor.

Preprocessor

The preprocessor performs initial tasks such as:

- Expanding macros

- Including header files

- Removing comments

It processes the source code before compilation begins.

Compiler

The compiler translates the preprocessed source code into **assembly code** or intermediate code. It also:

- Checks syntax and semantics

- Performs code optimizations

- Converts high-level code into low-level form

Assembler

The assembler is responsible for converting the assembly code - produced by the compiler - into machine code, which consists of binary instructions that a computer's CPU can directly execute. This machine code is often referred

to as object code and typically has the extension `.o` or `.obj`. Assemblers also manage symbol resolution, such as variable and function addresses, and assign memory locations for instructions and data. It bridges the gap between low-level symbolic representation and actual hardware-understandable instructions.

Linker

The linker combines:

- Multiple object files
- External library functions (e.g., `printf()`)

Produce a single executable file.

Loader

The loader is the final component of the language processing system. Once the executable file is created by the linker, the loader's job is to load this executable into the main memory (RAM) for execution. It sets up the runtime environment, including the stack, heap, and static memory segments. It also resolves dynamic linking for shared libraries if necessary. After preparation, the loader transfers control to the program's entry point, allowing the CPU to begin execution. The entire process is typically managed by the operating system

Summary Diagram
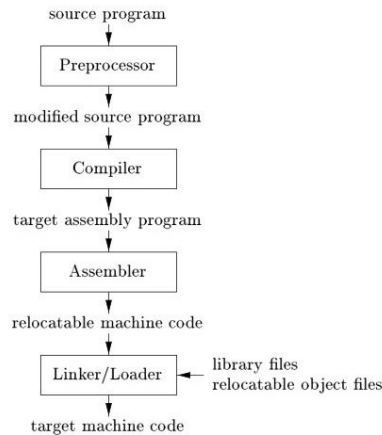
Figure 1.5: A language-processing system

1.4. Applications of Compilers

Compilers play an important role in many areas of computer science and software engineering. Let's look at some key uses.

1.4.1   Programming Language Implementation

To create a programming language, a compiler or interpreter must be developed. This allows developers to write code in that language and run it on machines.

1.4.2   Software Development

Compilers help software engineers:

- Catch syntax errors before running the program
- Optimize code for speed and memory

- Generate secure and portable executables

Most modern software (apps, games, OS) is built with the help of compilers.

1.4.3 Cross-Platform Development

Some compilers can create machine code that runs on a **different operating system or platform**. This is useful when building software for Windows, Linux, and macOS.

1.4.4 Compiler-Based Tools

Many tools are based on compiler principles, including:

- **Linters**: Check code style and formatting

- **Static analyzers**: Find bugs without running the program

- **IDEs (VSCode, Eclipse)**: Provide real-time error checking and code suggestions

These tools improve developer productivity and code quality.

1.4.5 Educational Purpose

Studying compilers helps students understand:

- How programming languages work internally

- How a source file becomes executable

- Core CS topics like automata, grammars, parsing

> **Bonus**
>
> Learning compiler design improves knowledge of **data structures**, **algorithms**, and **formal language theory** — useful for software engineering and competitive programming.

1.5. Problems and Exercises

1.5.1 Programming Problems

Problem 1: Write a program that will count the length of a string.

```
#include<stdio.h>
int main()
{
    char str[1000];
    scanf("%[^\n]s", str);
    int total_length = 0;
    for(int i = 0; str[i] != '\0'; i++)
    {
        total_length++;
    }
    printf("Total length: %d", total_length);
    return 0;
}
```

> **Sample Input and Output**
>
> | Input | Output |
> | --- | --- |
> | Hello Compiler World | Total length: 20 |

Problem 2:   Write a C program that will count the number of white spaces from a string.

```
#include <stdio.h>
int main()
{
    char str[1000];
    scanf("%[^\n]s", str);
    int white_space = 0;
    for(int i = 0; str[i] != '\0'; i++)
    {
        if(str[i] == ' ')
        {
            white_space++;
        }
    }
    printf("Total white space: %d", white_space);
    return 0;
}
```

Sample Input and Output

Input	Output
I love programming in C	Total white space: 4

Problem 3: Write a C program that will remove white spaces from a string.

```
#include<stdio.h>
int main()
{
    char str[1000];
    scanf("%[^\n]s", str);
    int j = 0;
    for(int i = 0; str[i] != '\0'; i++)
    {
        if(str[i] != ' ')
        {
            str[j] = str[i];
            j++;
        }
    }
    str[j] = '\0';
    printf("%s", str);
    return 0;
}
```

Sample Input and Output

Input	Output
Welcome to C programming	WelcometoCprogramming

1.5.2 Exercises

1. Write a C program that will count the number of vowels in a string.

2. Write a program to convert all lowercase letters in a string to uppercase.

3. Write a C program to reverse a string without using library functions.

4. Write a program to count the number of words in a sentence.

5. Write a C program to check whether a string is a palindrome or not.

Explore the codes in GitHub
Click or Scan the QR

Chapter 2

Phases of a Compiler

Many of us think compilers instantly convert code into machine language, but the process is much more complex. Instead of a single step, compilation happens in multiple stages—each with a specific job. These phases carefully analyze, optimize, and translate the code to ensure it runs correctly and efficiently.

Understanding these steps is key to building good compilers. In this chapter, we'll break down the six major phases of compilation and explore tools that help automate parts of the process. By the end, you'll see how raw source code gets transformed into a program your computer can execute.

2.1. The Six Core Phases of a Compiler

Compiling code is a multi-stage transformation process where each phase plays a crucial role in converting human-written programs into efficient machine code. This sophisticated translation occurs through six interconnected phases, logically grouped into two primary stages:

Analysis Phase (Frontend)

The analysis phase reads and understands the source code — it checks for errors, grammar, and meaning.

1. **Lexical Analysis:** Scans the source code and breaks it into tokens

— the smallest meaningful units such as keywords, identifiers, and operators.

2. **Syntax Analysis:** Checks the structure of the program using grammar rules and constructs a syntax tree or parse tree to represent the program's structure.

3. **Semantic Analysis:** Ensures the program is meaningful by checking things like type consistency, declared variables, and correct use of identifiers.

4. **Intermediate Code Generation:** Translates the analyzed source code into an intermediate representation that is easier to optimize and machine-independent.

Synthesis Phase (Backend)

The synthesis phase takes the analyzed code and produces optimized machine code ready for execution.

1. **Code Optimization:** Improves the intermediate code to make it more efficient without changing its meaning — for example, removing redundant operations or improving memory usage.

2. **Code Generation:** Converts the optimized code into actual machine instructions for the target system.

Throughout this pipeline, two critical components maintain consistency:

- **Symbol Table:** A key data structure that stores details about identifiers (variables, functions, constants) such as their types, scopes, and memory locations. It helps multiple phases coordinate correctly.

- **Error Handler:** Operates throughout all compiler phases to detect and report errors. It identifies lexical errors (e.g., invalid characters), syntax errors (e.g., missing brackets), and semantic errors (e.g., undeclared variables). Proper error handling ensures that the compiled program follows all rules of the language and makes debugging easier.
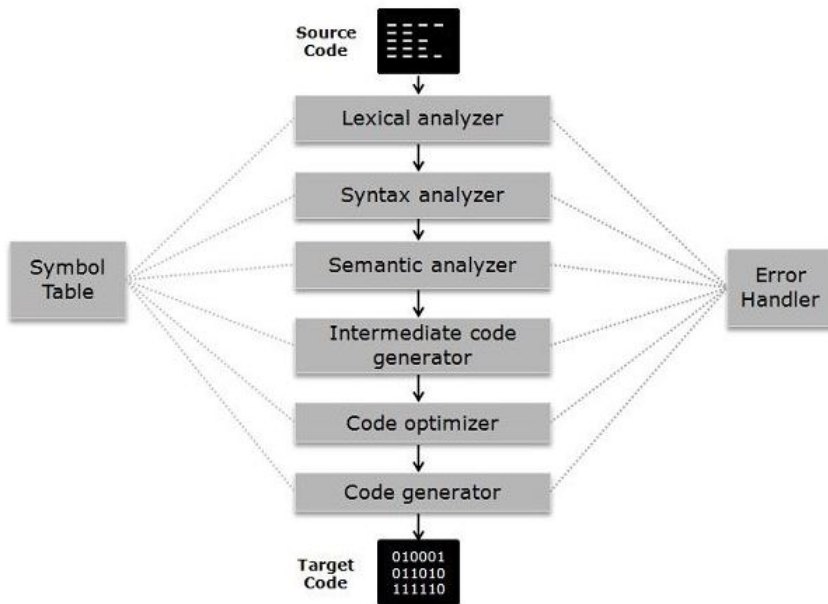
Figure 2.1: Six Phases of a Compiler

2.2. Lexical Analysis

Lexical Analysis is the first phase of a compiler. It acts as a translator between raw source code (written by a programmer) and the structure that later phases of the compiler can understand. This stage is often handled by a tool called a *lexer* or *scanner*.

In this phase, the compiler scans the source code character by character and groups them into tokens, which are the smallest meaningful units in a program, such as:

- Keywords: `int`, `while`, `return`
- Identifiers: `x`, `y`, `totalSum`

- Operators: +, =, *

- Constants: 100, 'A', 3.14

- Punctuation: ;, (,)

What happens in Lexical Analysis?

Input: The raw program text written by the programmer.

Process:

- Reads the input from left to right.

- Breaks the input into lexemes.

- Converts each lexeme into a token with type and value.

- Populates a symbol table with identifiers and literals.

Output: A stream of tokens used by the syntax analyzer.

Example:

Suppose, a source program contains this statement:

```
x = y * z + 5
```

The lexical analyzer converts this expression into a stream of tokens and updates the symbol table as follows:

Identifier Table

Token Name	Attribute Name
x	<id,1>
y	<id,2>
z	<id,3>

Operator Table

Token Name	Attribute Name
=	<op,1>
*	<op,2>
+	<op,3>

Constant Table

Token Name	Attribute Name
5	<c,1>

After lexical analysis, the statement `x = y * z + 5` would be represented as the following sequence of tokens (tokenization):

x	**=**	**y**	*	**z**	**+**	**5**
$\langle id, 1 \rangle$	$\langle op, 1 \rangle$	$\langle id, 2 \rangle$	$\langle op, 2 \rangle$	$\langle id, 3 \rangle$	$\langle op, 3 \rangle$	$\langle c, 1 \rangle$

Each token is passed to the syntax analyzer for further processing. The scanner also removes irrelevant elements for the next stage, like:

- Whitespace (spaces, tabs, newlines)
- Comments (`//this is a comment` or `/* comment */`)

2.3. Syntax Analysis

The second phase of the compiler is **syntax analysis** or **parsing**. It is the compiler's "grammar checker." It takes the list of tokens from the lexical analyzer and verifies whether they form valid, meaningful statements according to the rules of the programming language.

Builds a syntax tree, enforcing precedence rules and representing the code's

hierarchical structure. The parser rejects any statement that breaks the grammatical rules of the language by raising a **syntax error**.
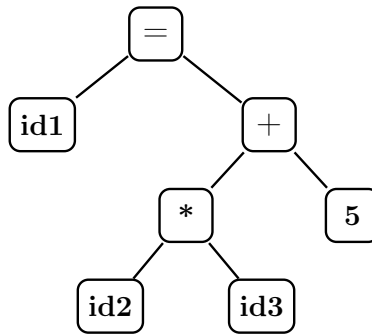
Figure2.2: Syntax Tree or Parse Tree

2.4. Semantic Analysis

The semantic analyzer checks whether a program actually makes sense, beyond just following grammar rules. It looks at the syntax tree and symbol table to make sure everything follows the meaning and logic of the programming language. This is the phase where the compiler catches errors like:

- using a variable before declaring it,

- calling a function with the wrong number of arguments, or

- trying to add a number to a piece of text.

An interesting part of semantic analysis is that it can also fix small mistakes automatically. For example, if you add an integer and a float, the compiler

will convert the integer to a float on its own. This is called **implicit type conversion**.
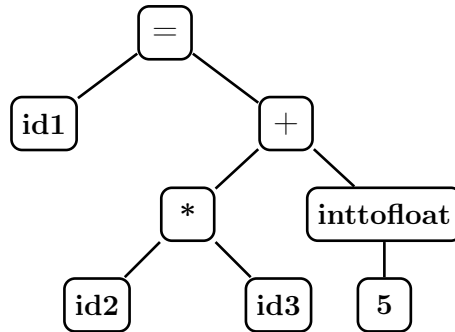
Figure 2.3: Syntax Tree in Semantic Analysis

2.5. Intermediate Code Generation

After the Semantic Analysis phase, the compiler converts the source program into an *intermediate representation* (IR). This IR is a simplified, machine-friendly form of the code that is:

1. easy to generate from the source code, and
2. easy to translate into actual machine code.

A common IR is **Three-Address Code (3AC)**, where each instruction has at most three operands (variables, constants, or temporaries) and one operator.

The Three Address Code derived from the syntax tree in Figure 2.3 is:

```
t1 = inttofloat(5)
t2 = id2 * id3
t3 = t1 + t2
id1 = t3
```
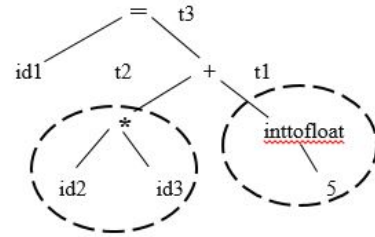

Figure 2.4: Three Address Code Generation

2.6. Code Optimization

The code optimization phase improves the intermediate code by making it faster, shorter, and more efficient without changing its output. The compiler analyzes the code and removes any unnecessary steps while keeping the logic correct. For example, if a number like 5 is being converted to a floating-point value (5.0) and used only once, the optimizer replaces it directly with 5.0, skipping the extra conversion. It also looks for temporary variables (like t1) that are used only once, and replaces them with the actual expression, reducing extra storage and simplifying the code.

In our case, the original code is optimized to:

```
t1 = id2 * id3
id1 = t1 + 5.0
```

This makes the program shorter and faster, helping the computer finish its task more quickly and efficiently.

2.7. Code Generation

In the final phase of compilation, the **code generator** translates the optimized intermediate code into actual machine instructions that a computer can execute. This involves assigning variables to CPU **registers** or **memory locations**, selecting the correct machine-level arithmetic instructions (like addition or multiplication), and directly embedding constants into the code when needed.

For example, the optimized code from the code optimization phase can be translated into machine-level instructions using registers such as R1 and R2:

```
LDF    R1, id2
LDF    R2, id3
MULF   R1, R1, R2
ADDF   R1, R1, #5.0
STF    id1, R1
```

Here, **LDF** loads floating-point values into registers, **MULF** & **ADDF** perform multiplication and addition, and **STF** stores the final result. This phase ensures that the machine code is efficient, compact, and performs the same computations as described in the original source program.

2.8. Errors in Compiler

Compilers detect errors during different phases of translation—from scanning source code to generating the final output. These errors can appear in *lexical analysis*, *syntax parsing*, *semantic checking*, *optimization*, or *code generation*. Each phase catches specific mistakes, helping programmers fix problems step by step.
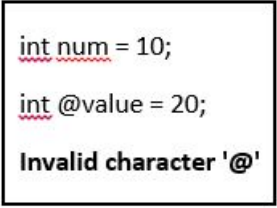
Common types of errors are described below:

2.8.1 Lexical Error

A lexical error occurs during the *lexical analysis phase* (also called scanning), where the compiler breaks the source code into tokens. If the scanner finds any invalid sequence of characters that doesn't match any valid token in the language, it reports a lexical error.

Common causes of lexical errors:

- Invalid characters (e.g., @, $ in C language)
- Unclosed strings (e.g., "Hello world)
- Malformed literals (e.g., 3.14.15, 'abc' in C)
- Invalid tokens (e.g., 123abc)
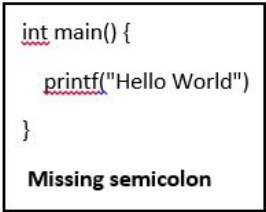
Figure 2.5: Lexical Error

2.8.2 Syntax Error

A syntax error occurs when code violates the grammatical rules of the programming language, making it impossible for the compiler or interpreter to parse the code.

Common causes of syntax errors:

- Missing symbols (semicolon, comma, etc.)
- Mismatched parentheses or brackets
- Incomplete statement

Figure 2.6: Syntax Error

2.8.3 Semantic Error

A semantic error occurs when code follows correct syntax (grammar) but violates logical rules, leading to unintended behavior.

Common causes of semantic errors:

- Data type mismatch (e.g., `"5" + 3`)
- Uninitialized variables.
- Array overflow
- Using float in array index

Figure 2.7: Semantic Error

2.8.4   Logical Error

A logical error occurs when a program runs without crashes (no syntax or runtime errors) but produces incorrect results due to incorrect logic in the code.

Common causes of logical errors:

- Incorrect formula (e.g., `average = sum / 2` instead of `sum / n`)
- Wrong condition in if-else or loop
- Off-by-one error (e.g., `for (i = 0; i <= 10; i++)` instead of `i < 10`)
- Division by zero

Figure 2.8: Logical Error

2.8.5   Why Your Code Crashed: An Error Guide!!

A simple code example is provided below to help identify and understand common types of programming errors.
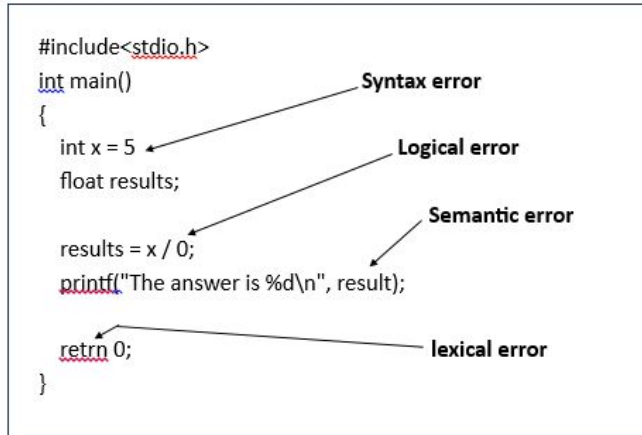
Here, we observe different types of errors:

- **Syntax Error:** `Int x = 5` — missing semicolon.

- **Logical Error:** `results = x / 0;` — division by zero.

- **Semantic Error:** `result` — used without being defined.

- **Lexical Error:** `retrn 0;` — misspelled keyword or invalid token.

2.9. Problems and Exercises

2.9.1   Programming Problems

Problem 1: Take Multiple Lines as Input and Print as Output

```c
#include <stdio.h>
int main() {
    char s[1000];
    printf("Enter the input: \n");
    scanf("%[^~]", s);
    printf("output: \n");
    printf("%s", s);
    return 0;
}
```

Sample Input and Output	
Input	**Output**
I am a student.	I am a student.
I love C.	I love C.
I also love AI.~	I also love AI.

Problem 2: Extract Single Line Comment

```c
#include <stdio.h>
int main() {
    char line[1000];
    int i = 0;
    printf("Enter line:\n");
    if (scanf("%[^\n]", line) == 1) {
        scanf("%*c");
    }
    while (line[i] != '\0') {
        if (line[i] == '/' && line[i + 1] == '/') {
            printf("Single line comment: %s\n", &line[i]);
            return 0;
        }
        i++;
    }
    return 0;
}
```

Sample Input and Output

Input	Output
int a = 5; // this is a variable	Single line comment: // this is a variable

Problem 3: Count Number of Lines

```c
#include <stdio.h>
int main() {
    char line[1000];
    int count = 0;
    printf("Enter multiple lines:\n");
    while (1) {
        if (scanf("%[^\n]", line)==1) {
            scanf("%*c");
            count++;
        }
        else {
            scanf("%*c");
            break;
        }
    }
    printf("Total Lines: %d\n", count);
    return 0;
}
```

Sample Input and Output

Input	Output
Hello World	Total Lines: 3
This is line two	
This is line three	

2.9.2 Exercises

Statement-Based Analytical Exercises

- Lexical Analysis

- Syntax Analysis

- Semantic Analysis

- Intermediate Code Generation

- Code Optimization

- Code Generation

1. For the statement `x = a*b + c*d + e/f + 5;`, analyze how each phase of the compiler processes it:

2. For the statement `sum = a + b - c;`, analyze it through all six compiler phases.

3. Analyze the statement `area = 3.14 * r * r;` using the six phases of a compiler.

4. Analyze `temp = (x + y) / 2;` through each compiler phase.

5. Analyze the boolean assignment `flag = (count > 10) && (status == 1);` through the six compiler phases.

Programming Problems on Compiler Phases

1. **Lexical Analyzer Simulator** Write a C program that reads a line of code and prints its tokens:

   Input: `int a = b + 5;`
   Output:

   ```
   Keyword: int
   Identifier: a
   Operator: =
   ```

```
Identifier: b
Operator: +
Number: 5
```

2. **Symbol Table Generator** Write a program that takes multiple variable declarations and builds a symbol table.
   Input: `int x; float y; char z;`
   Output:

   ```
   Name    | Type
   ---------------
   x       | int
   y       | float
   z       | char
   ```

3. **Intermediate Code Generator** Create a program that converts an infix expression to 3-address code.
   Input: `a = b + c * d;`
   Output:

   ```
   t1 = c * d
   t2 = b + t1
   a = t2
   ```

4. **Type Checking Simulation** Write a program that detects type mismatch in simple assignments.
   Input: `int a; float b; a = b + 2.3;`
   Output: Type Mismatch Error: int = float

Explore the codes in GitHub
Click or Scan the QR

Chapter 3

Lexical Analysis & Context-Free Grammar

3.1. Lexemes, Tokens and Patterns

3.1.1 Lexeme

A **lexeme** is the smallest sequence of characters that is meaningful in a programming language. For example, in the statement `int a = 5;` the lexemes are: `int`, `a`, `=`, `5` and `;`.

3.1.2 Token and Types of Lexical Tokens

A **token** is a set or category of lexemes. When the lexical analyzer reads the source code, it groups sequences of characters into tokens based on the rules of the language.

Common Types of Lexical Tokens:

1. **Keywords**
   Reserved words that are predefined in the language and cannot be used as identifiers.
   Examples: `int`, `while`, `if`, `return`

2. **Identifiers**
   Names used for variables, functions, arrays, classes, etc.
   Examples: `x`, `main`, `totalSum`

3. **Operators**
   Symbols used to perform operations.
   Examples: `+`, `-`, `*`, `/`, `=`

4. **Literals (Constants)**
   Values that are fixed and used directly in the code.
   Examples: `10`, `3.14`, `'A'`, `"Hello"`

5. **Punctuation (Separators or Delimiters)**
   Symbols that define structure or separate statements.
   Examples: `;`, `,`, `{`, `}`, `(`, `)`

6. **Comments**
   Not executed; used to add notes to the code. Usually ignored by the compiler.
   Examples: `// single-line`, `/* multi-line */`

NOTE

Tokens are generated during the lexical analysis phase and passed on to the parser for syntax analysis.

3.1.3   Pattern

A **pattern** is a rule that describes the form a token may take. Typically defined using regular expressions.

3.1.4   Examples of Tokens in Programming Language

In programming languages, a line of code is broken down into multiple tokens by the lexical analyzer. Each token belongs to a specific type based on its role in the syntax. Let us consider the following C statement:

```
int x = a * b + 10 + 3.5;
```

The tokens extracted from the statement and their types are shown below:

Lexeme	Token Type	Description
int	Keyword	Data type declaration
x	Identifier	Variable name
=	Operator	Assignment operator
a	Identifier	Variable
*	Operator	Multiplication operator
b	Identifier	Variable
+	Operator	Addition operator
10	Literal	Integer constant
+	Operator	Another addition operator
3.5	Literal	Floating-point constant
;	Punctuation	Statement terminator

This example illustrates how the lexical analyzer identifies and classifies each part of the statement during the lexical analysis phase of the compiler.

3.2. Lexical Errors

Lexical errors occur when the lexical analyzer encounters invalid input that it cannot recognize as a valid token.

These errors are detected during the first phase of the compiler — lexical analysis. If a character or sequence does not match any defined pattern for tokens, it results in a lexical error.

Types of Lexical Errors

1. **Illegal Characters:**
   Characters that are not allowed in the source code.

Example: `@`, `#`, `~`

2. **Invalid Identifiers:**
   Identifiers that start with a digit or include invalid symbols.
   Example: `123abc`, `total$value`

3. **Unterminated String Literals:**
   A string that does not have a closing quotation mark.
   Example: `"Hello world`

4. **Missing Spaces Between Tokens:**
   When multiple tokens are written together without proper separation.
   Example: `inta=5;` instead of `int a = 5;`

5. **Invalid Escape Sequences:**
   Use of unsupported escape characters in strings.
   Example: `"Hello\qWorld"`

NOTE

Lexical errors are the earliest errors detected by the compiler. These must be corrected before the parser can analyze the program's structure.

3.3. Error Recovery Strategies

When a lexical error occurs, the compiler attempts to handle it using specific strategies, instead of stopping immediately. This allows the compiler to continue processing the rest of the code and detect additional errors.

Common Error Recovery Strategies:

1. **Panic Mode Recovery**
   The compiler skips input symbols until it finds a known synchronizing token (like `;` or `}`).
   Example: If an illegal symbol appears, skip characters until the end of

the line.

2. **Deletion**
   Delete the invalid character and continue scanning.
   Example: `#x = 5;` becomes `x = 5;` after removing `#`.

3. **Insertion**
   Assume a character is missing and insert the expected one.
   Example: `inta = 5;` becomes `int a = 5;` after inserting a space.

4. **Substitution**
   Replace an invalid character with a valid one.
   Example: Replace `@` with `=` in `x @ 5`.

5. **Transposition**
   Swap adjacent characters to correct the token.
   Example: `nto` becomes `not`.

NOTE

Error recovery strategies make compilers more robust by allowing them to continue processing even when small mistakes are found in the source code.

3.4. Context-Free Grammar (CFG)

Context-Free Grammar (CFG) is a powerful formal system used to describe the syntax of programming languages. It plays a critical role in the compiler's parsing phase. CFG defines how strings in a language can be formed using rules known as production rules.

A CFG can generate all possible valid strings in a language, ensuring that the structure of code follows the grammar.

A CFG is represented as:

$$G = (V, \Sigma, P, S)$$

Where:

- V: Set of non-terminal symbols

- Σ: Set of terminal symbols

- P: Set of production rules

- S: Start symbol

Example 1: Arithmetic Expressions

```
E → E + E | E * E | (E) | id
```

This grammar can generate strings such as:

- `id + id`

- `id * (id + id)`

- `(id + id) * id`

Example 2: Assignment Statements

```
S → id = E
E → E + E | id
```

3.4.1 Use of CFG in Syntax Analysis

CFG is used by the parser to determine if the structure of a program is valid. It helps to:

- Construct parse trees

- Detect syntax errors

- Guide semantic analysis

- Generate intermediate code

CFGs are used in designing various types of parsers, including:

- LL(1) parsers

- LR(0), SLR, LALR parsers
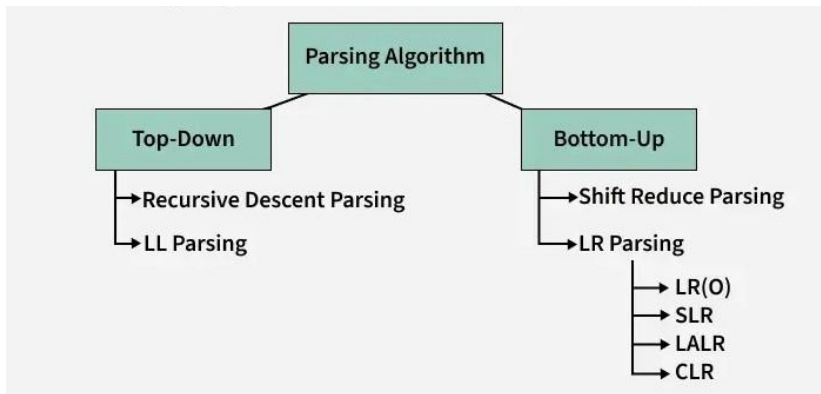
Figure 3.1: Syntax Analysis

3.4.2   Grammar Symbols: Terminal & Non-Terminal

Terminal Symbols (Σ) are the tokens from the source code.

Examples include:

- +, *, (,), id, =

Non-Terminal Symbols (V) are abstract symbols used to define grammar structure.

Examples include:

- E, S, T, F

Terminals appear in the final code, while non-terminals define how code is structured.

3.4.3   Parse Trees and Derivations

A parse tree shows how a sentence is derived from the start symbol using grammar rules.

There are two types of derivations:

- **Leftmost Derivation:** Leftmost non-terminal is replaced first

- **Rightmost Derivation:** Rightmost non-terminal is replaced first

Example: Grammar:

```
E → E + E | id
```

String: `id + id`

Leftmost Derivation:

```
E → E + E
E → id + E
E → id + id
```

Parse Tree:
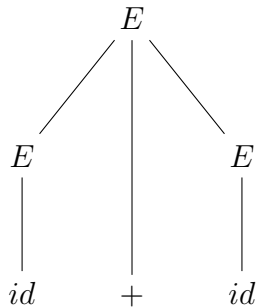
> **NOTE**
>
> Parse trees are useful for semantic analysis and intermediate code generation.

3.5. Ambiguity in Grammar

A grammar is said to be **ambiguous** if there exists at least one string with more than one distinct parse tree or derivation. This causes confusion in syntax analysis and is undesirable in programming languages.

3.5.1   Ambiguity

Let G be a context-free grammar. G is **ambiguous** if there exists a string $w \in L(G)$ such that w has two or more different leftmost derivations, rightmost derivations, or parse trees.

3.5.2   Leftmost and Rightmost Derivations

In a context-free grammar (CFG), derivations are sequences of production rule applications used to generate strings from the start symbol. Two common types of derivations are leftmost and rightmost derivations.

Leftmost Derivation: In this method, at each step, the leftmost non-terminal in the current string is replaced first using an applicable production rule. This approach proceeds from left to right, and every derivation step focuses only on the first available non-terminal on the left. Leftmost derivations are often used in top-down parsing strategies, such as recursive descent parsers.

Rightmost Derivation: In contrast, this method replaces the rightmost non-terminal in the string at every step. It progresses from right to left by targeting the last non-terminal symbol. Rightmost derivations are the basis for bottom-up parsing, like shift-reduce parsers and LR parsers. Despite producing the same final string, the steps of derivation and intermediate forms may differ from the leftmost version.

Grammar:

E → E + E | E * E | (E) | id

Input String: `id + id * id` **Leftmost Derivation:**

```
→ E
→ E + E                        [E → E + E]
→ id + E                       [E → id]
→ id + E * E                   [E → E * E]
→ id + id * E                  [E → id]
→ id + id * id                 [E → id]
```

Rightmost Derivation:

```
→ E
→ E + E                        [E → E + E]
→ E + E * E                    [E → E * E]
→ E + id * E                   [E → id]
→ E + id * id                  [E → id]
→ id + id * id                 [E → id]
```

3.5.3 Ambiguous Grammar Examples

Ambiguity in Grammar

Grammar:
E → E + E | E * E | id
Input String: id + id * id
Leftmost Derivation (interpreted as (id + id) * id):
→   E
→   E + E
→   id + E
→   id + E * E
→   id + id * id
Rightmost Derivation (interpreted as id + (id * id)):
→   E
→   E + E
→   E + E * E
→   E + id * E
→   E + id * id
→   id + id * id
Since the same string is derived with two different parse trees, the grammar is ambiguous.

Parse Trees:
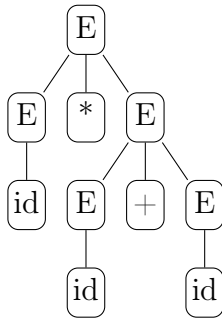
Left Derivation: **Right Derivation:**
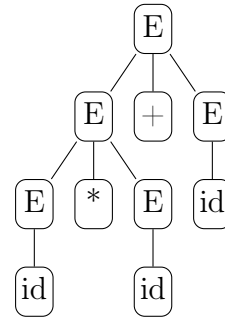
> **NOTE**
>
> This grammar is ambiguous because the string `id + id * id` has two
> valid parse trees that group operations differently.

3.5.4   Eliminating Ambiguity

Ambiguity in grammar makes parsing difficult, as one string can be in-
terpreted in multiple ways (multiple parse trees). To avoid confusion in
compiler design, we eliminate ambiguity by rewriting the grammar to:

- Operator **precedence**: * before +

- Operator **associativity**: both are left-associative

Original (Ambiguous) Grammar:

E → E + E | E * E | id

New (Unambiguous) Grammar:

```
E → E + T | T
T → T * F | F
F → id
```

Derivation of `id + id * id`:

```
→ E
→ E + T
→ T + T
→ F + T
→ id + T
→ id + T * F
→ id + F * F
→ id + id * id
```

> **NOTE**
>
> This derivation ensures that `*` has higher precedence than `+`, eliminating the ambiguity present in the original grammar.

3.6. Problems and Exercises

3.6.1   Programming Problems

Problem 1: Write a C program that will tokenize an input string **without using** the `strtok()` function.

```c
#include<stdio.h>
int main()
{
    char s[1000];
    char st[1000];
    scanf("%[^\n]s",s);
    int j=0;
    for(int i=0;s[i]!='\0';i++)
    {
        if(s[i]>='a' && s[i]<='z')
        {
            st[j]=s[i];
            j++;
        }
        else if(s[i]>='0' && s[i]<='9')
        {
            if(s[i+1]>='0' && s[i+1]<='9')
            {
                printf("%c",s[i]);
            }
            else
            {
                printf("%c\n",s[i]);
            }
        }
        else if(s[i]==' ')
        {
            st[j]='\0';
            printf("%s\n",st);
            j=0;
```

```
        }
        else
        {
            printf("%c\n",s[i]);
        }
    }
    st[j]='\0';
    printf("%s\n",st);
    return 0;
}
```

Sample Input and Output

Input	Output
An12 example, test input 88 end.	an
	12
	example
	,
	test
	input
	88
	end
	.

Problem 2: Write a C program that will identify the articles ("a", "an", "the") from a given input string and count the number of articles.

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int is_article(char word[]) {
    // Convert word to lowercase
    for(int i = 0; word[i]; i++){
        word[i] = tolower(word[i]);
    }
    return (strcmp(word, "a") == 0 || strcmp(word, "an")
        == 0 || strcmp(word, "the") == 0);
}

int main() {
    char str[1000];
    char word[50];
    int i = 0, j = 0, count = 0;

    printf("Enter a sentence:\n");
    fgets(str, sizeof(str), stdin);

    while(str[i] != '\0') {
        if(str[i] != ' ' && str[i] != '\n') {
            word[j++] = str[i];
        } else {
            word[j] = '\0';
            if(is_article(word)) {
                count++;
                printf("Found article: %s\n", word);
            }
            j = 0;
        }
        i++;
    }
```

```
    // Check last word
    word[j] = '\0';
    if(is_article(word)) {
        count++;
        printf("Found article: %s\n", word);
    }

    printf("Total number of articles: %d\n", count);
    return 0;
}
```

Sample Input and Output

Input	Output
The cat saw a mouse in the garden and an owl on the tree.	Found article: The
	Found article: a
	Found article: the
	Found article: an
	Found article: the
	Total number of articles: 5

Problem 3: Write a C++ program to test whether a given identifier is valid or not, and also check if the identifier is a C++ keyword.

```cpp
#include <bits/stdc++.h>
using namespace std;

string keywords[] = {
 "alignas", "alignof", "and", "asm", "auto", "bool", "break",
     "case", "catch",
 "char", "class", "const", "const_cast", "continue", "
    decltype", "default",
 "delete", "do", "double", "dynamic_cast", "else", "enum", "
    explicit", "export",
 "extern", "false", "float", "for", "friend", "goto", "if", "
    inline", "int",
 "long", "mutable", "namespace", "new", "noexcept", "nullptr"
    , "operator", "private",
 "protected", "public", "register", "reinterpret_cast", "
    return", "short", "signed",
 "sizeof", "static", "static_assert", "static_cast", "struct"
    , "switch", "template",
 "this", "thread_local", "throw", "true", "try", "typedef", "
    typeid", "typename",
 "union", "unsigned", "using", "virtual", "void", "volatile",
     "wchar_t", "while"
};

bool isLetter(char ch) {
 return (ch >= 'A' && ch <= 'Z') || (ch >= 'a' && ch <= 'z');
}

bool isDigit(char ch) {
 return (ch >= '0' && ch <= '9');
}

bool isKeyword(const string &str) {
 for (const string &kw : keywords) {
   if (kw == str)
     return true;
 }
 return false;
}
```

```
bool isValidIdentifier(const string &str) {
 if (str.empty()) return false;
 if (!(isLetter(str[0]) || str[0] == '_'))
   return false;
 for (size_t i = 1; i < str.length(); i++) {
   if (!(isLetter(str[i]) || isDigit(str[i]) || str[i] == '_'
      ))
     return false;
 }
 return true;
}

int main() {
 string str;
 cout << "Enter an identifier: ";
 cin >> str;
 if (isKeyword(str))
   cout << "'" << str << "' is a keyword (Not a valid
      identifier)." << endl;
 else if (isValidIdentifier(str))
   cout << "'" << str << "' is a valid identifier." << endl;
 else
   cout << "'" << str << "' is NOT a valid identifier." <<
      endl;
 return 0;
}
```

Sample Input and Output

Input	Output
int	'int' is a keyword (Not a valid identifier).
my_var1	'my_var1' is a valid identifier.
1stVar	'1stVar' is NOT a valid identifier.
_hidden	'_hidden' is a valid identifier.

3.6.2 Exercises

Lexical Analysis Exercises

Academic Exercises

Q1. Define the role of a lexical analyzer in the compilation process.

Q2. Differentiate between lexeme, token, and pattern with examples.

Q3. Write regular expressions for the following:

  (a) An identifier that starts with a letter followed by any number of letters or digits.

  (b) A real number with optional fractional and exponential parts.

Q4. Construct a finite automaton for recognizing a simple identifier.

Q5. Explain how lexical errors are handled during compilation.

Programming Exercises

P1. Write a program in Python that uses regular expressions to extract all identifiers from a given code snippet.

P2. Implement a lexical analyzer for arithmetic expressions (supporting +, -, *, /, numbers, and variables).

P3. Simulate a DFA in code that recognizes binary numbers divisible by 3.

P4. Write a C/C++ program to tokenize a C-style source file and count the number of keywords, identifiers, and literals.

Context-Free Grammar (CFG) Exercises

Academic Exercises

Q6. Given the following CFG:

$$S \rightarrow aSb \mid \varepsilon$$

Show the derivation steps for the string `aabb`.

Q7. Construct a CFG that generates palindromes over the alphabet {a, b}.

Q8. Eliminate left recursion from the following grammar:

$$A \rightarrow A\alpha \mid \beta$$

Q9. Convert the following grammar to Chomsky Normal Form:

$$S \rightarrow ASA \mid aBA \rightarrow B \mid S \mid aB \rightarrow b$$

Programming Exercises

P5. Write a Python program using the 'nltk' or 'lark' library to parse strings using a given CFG.

P6. Implement a parser that takes a CFG and checks whether a given string is accepted using the CYK algorithm.

P7. Create a program to convert a simple CFG into its equivalent left-factored form.

P8. Simulate derivations of a CFG and print the parse tree using any programming language.

Explore the codes in GitHub
Click or Scan the QR

Chapter 4

Regular Expressions

4.1. Basics of Regular Expressions

A **Regular Expression (RE)** is a symbolic representation used to describe
patterns in strings. In compiler design, regular expressions are used to de-
fine the lexical rules for tokens such as identifiers, keywords, numbers, and
operators.

- Provides a compact and expressive way to define lexical tokens.

- Forms the basis for creating lexical analyzers.

- Can be automatically translated to Finite Automata.

Regular expressions are widely used in tools like `Lex` and `Flex`, and also in
programming languages such as Python, Perl, and JavaScript.

Alphabet and Strings

- An **alphabet** (Σ) is a finite set of characters or symbols. Example:
  $\Sigma = \{a, b, 0, 1\}$.

- A **string** is a finite sequence of symbols from an alphabet. Example:
  "ab", "101".

- The **empty string** ε contains no symbols.

- The set of all strings (including ε) over Σ is denoted by Σ^*.

4.2. Operators in Regular Expressions

Union (|)

- Syntax: `r1 | r2`

- Matches either expression `r1` or `r2`

- Example: `a|b` matches "a" or "b"

Concatenation

- Syntax: `r1 r2`

- Matches strings where `r1` is followed by `r2`

- Example: `ab` matches "ab"

Kleene Star ($*$)

- Syntax: `r*`

- Matches zero or more repetitions of `r`

- Example: `a*` matches $\{\varepsilon, a, aa, aaa, \dots\}$

Precedence and Parentheses

Operator precedence (from highest to lowest):

Operator	Description
*	Kleene star (zero or more)
Concatenation	Implied when expressions are next to each other
|	Union (alternation)

> **Examples of Grouping**
>
> - `a|bc` means `a` or `bc`
> - `(a|b)c` means `ac` or `bc`
> - `(ab)*` means ε, `ab`, `abab`, `ababab`, ...

Common Patterns

- `[0-9]` → Any digit

- `[a-z,A-Z]` → Any alphabet character

- `[a-z][0-9]` → A letter followed by digits

- `(a|b)*abb` → Any string of a's and b's ending with "abb"

Usage in Lexical Analysis

- `keyword` → `if|else|while|return`

- `id` → `[a-z,A-Z][a-z,A-Z,0-9]`

- `number` → `[0-9]+(\.[0-9]+)`

4.3. Language of Regular Expressions

A **language** is a set of strings over an alphabet. Regular expressions define **regular languages**, which can be recognized by Finite Automata.

Regular Expressions Basics

- `a` → $\{a\}$

- `a|b` → $\{a, b\}$

- `ab` → $\{ab\}$

- a^* → $\{\varepsilon, a, aa, aaa, \dots\}$

- $a^+ \rightarrow \{a, aa, aaa, \dots\}$

- $a^{2+} \rightarrow \{aa, aaa, aaaa, \dots\}$

- $a^? \rightarrow \{\varepsilon, a\}$

Regular Expressions Basics

Let the alphabet be $\Sigma = \{a, b\}$, which means it contains only the letters a and b.

- $(ab)^* = \{\varepsilon, ab, abab, ababab, \dots\}$
  Zero or more repetitions of the string ab.

- $a \cup b = \{a, b\}$
  Union of the single letters a and b.

- $(a \cup b)^* =$ All strings (including empty string) made using any number of a's and b's in any order.
  Example: $\{\varepsilon, a, b, aa, ab, ba, bb, aab, \dots\}$

- $(a^*b^*)^* = (ab^*)^* =$ All strings of a's and b's in any order, possibly grouped.
  Example: $\{\varepsilon, a, b, ab, abb, aabb, abbb, aaab, \dots\}$

- $a^*b^* = \{a^i b^j \mid i \geq 0, j \geq 0\}$
  All strings with zero or more a's followed by zero or more b's. Example: $\{\varepsilon, a, b, aa, ab, aab, abb, aabb, \dots\}$

NOTE

Regular expressions are closed under union, concatenation, and Kleene star. They can also be converted into equivalent NFAs or DFAs.

4.3.1 Language to Regular Expression Conversion

Let $\Sigma = \{a, b\}$ – all expressions over this alphabet.

- Strings with **exactly one** a:

$$b^* a b^*$$

- Strings with **exactly two** a's:

$$b^* a b^* a b^*$$

- Strings with **one or more** a's:

$$(b^* a b^*)^* \quad \text{or} \quad (a \cup b)^* a (a \cup b)^*$$

- Strings with **even number of** a's:

$$(b^* a b^* a b^*)^*$$

- Strings with **even number of** a's **and exactly one** b:

$$(aa)^* b (aa)^* \cup (aa)^* a b (aa)^* a$$

- Strings with **odd number of** a's:

$$(b^* a b^* a b^*)^* b^* a b^*$$

- Strings that **don't contain** aa:

$$(b \cup ab)^* (\epsilon \cup a)$$

- "Set of all strings having at least one ab"

$$(ab)^+$$

- "Set of all strings having even number of aa"

$$(aa)^*$$

- "Set of all strings having odd number of bb"

$$b(bb)^*$$

- "Set of all strings having even number of aa and even number of bb"

$$(aa)^*(bb)^*$$

- "Set of all strings having zero or more instances of a or b starting with aa"

$$(aa)(a \mid b)^*$$

- "Set of all strings having zero or more instances of a or b ending with bb"

$$(a \mid b)^*(bb)$$

- "Set of all strings having zero or more instances of a or b starting with aa and ending with bb"

$$(aa)(a \mid b)^*(bb)$$

4.3.2 Regular Expression to Language

Let $\Sigma = \{a, b\}$ – all expressions over this alphabet.

1. $a \mid b$
   Denotes the set $\{a, b\}$ (either a or b)

2. $(a \mid b)(a \mid b)$
   Denotes $\{aa, ab, ba, bb\}$ (all length-2 strings over $\{a, b\}$)

3. a^*
   Denotes $\{\epsilon, a, aa, aaa, \ldots\}$ (all strings of zero or more a's)

4. $(a \mid b)^*$ or $(a^* \mid b^*)^*$
   Denotes all strings over $\{a, b\}$ (including ϵ)

5. $a \mid a^*b$
   Denotes $\{a\} \cup \{$all strings $a^n b$ for $n \geq 0\}$

6. $a(a \mid b)^*a$
   All strings over $\{a, b\}$ that begin and end with a

7. $(a \mid b)^*a(a \mid b)(a \mid b)$
   Strings with a as the third character from the end

8. $(aa)^*$
   Strings with even number of a's (including ϵ)

9. $(a \cup b)^4$
   All length-4 strings over $\{a, b\}$

10. $((a \cup b)^4)^*$
    Strings whose length is divisible by 4

11. $(aa)^* \cap ((a \cup b)^4)^*$
    Strings with both: even number of a's AND length divisible by 4

NOTE

- Alternative notation:
  - $\cup$ can be written as $\mid$ in regex
  - $L_1 \cap L_2$ requires both conditions
  - r^* always includes ϵ

4.4. Limitations of Regular Expressions

Regular expressions are useful for defining patterns in strings, especially for lexical analysis in compilers. However, they cannot describe all kinds of patterns, especially when the structure of the language is complex or nested.

4.4.1   What They Can't Express

Regular expressions fail in the following scenarios:

- **Matching nested structures:** Regular expressions cannot handle balanced parentheses like (), (()), (()()), etc.

- **Counting equal numbers:** They cannot express languages where the number of one symbol must match another, such as $\{a^n b^n \mid n \geq 0\}$.

- **Palindromes:** Strings that read the same forwards and backwards, like abba, abcba, cannot be described by regular expressions.

- **Repeated patterns:** Strings like ww, where the second half exactly repeats the first (e.g., abab, xyzxyz) are not regular.

- **Nested programming blocks:** Structures like nested if statements or braces {{...}} in code are not possible using regular expressions.

- **Recursive structures:** Regular expressions cannot define recursive language rules like if...else if...else... or nested HTML/XML tags.

The reason is that regular expressions are based on **finite automata**, which do not have memory (like a stack) to keep track of nested or recursive structures.

4.4.2   Context-Free Language Examples

Context-Free Grammars (CFGs) can handle these complex structures using recursive rules. These languages can be recognized by **pushdown au-**

tomata (PDA), which have a stack for memory.

> ### Examples of Context-Free (Non-Regular) Languages
>
> - $L = \{a^n b^n \mid n \geq 0\}$
>   Equal number of a's followed by equal number of b's.
> - Balanced parentheses:
>   (), (()), (()(())), etc.
> - Arithmetic expressions with nested operations:
>   (a + b), ((a + b) * c), a * (b + (c - d))
> - Palindromes:
>   abba, abcba, radar, level

4.5. Problems and Exercises

4.5.1 Programming Problems

Problem 1: Match All a-b Strings – (a|b)*

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);

    int valid = 1;
    for (int i = 0; i < strlen(str); i++) {
        if (str[i] != 'a' && str[i] != 'b') {
            valid = 0;
            break;
        }
    }

    if (valid)
        printf("Accepted\n");
    else
        printf("Rejected\n");

    return 0;
}
```

Sample Input and Output

Input	Output
aababb	Accepted
abc	Rejected

Problem 2: Match Grouped a's and b's − (a*|b*)*

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);

    int valid = 1;
    for (int i = 0; i < strlen(str); i++) {
        if (str[i] != 'a' && str[i] != 'b') {
            valid = 0;
            break;
        }
    }

    if (valid)
        printf("Accepted\n");
    else
        printf("Rejected\n");

    return 0;
}
```

Sample Input and Output

Input	Output
aaaaabbbbb	Accepted
ababab	Accepted
abc	Rejected

Problem 3: Match Even-Length ab Pairs – (ab)*

```c
#include <stdio.h>
#include <string.h>

int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);

    int len = strlen(str);
    int valid = 1;

    if (len % 2 != 0) valid = 0;
    for (int i = 0; i < len && valid; i += 2) {
        if (str[i] != 'a' || str[i+1] != 'b') {
            valid = 0;
            break;
        }
    }

    if (valid)
        printf("Accepted\n");
    else
        printf("Rejected\n");

    return 0;
}
```

Sample Input and Output

Input	Output
abab	Accepted
aba	Rejected
ababb	Rejected

4.5.2   Exercises

1. Write a C program to check whether a given string is a binary string (i.e., consists only of 0 and 1). *Pattern: (0|1)**

2. Accept strings that contain an even number of a characters (e.g., aa, aabb, bbaabb). *Hint: Use a counter to track number of a's.*

3. Accept strings that match the pattern: a*b*c* (e.g., aaabbbccc, abc, aabbcc, but not acb, cba).

4. Check if a string is a valid C identifier. Conditions:

   - Starts with a letter or underscore (_)

   - Followed by letters, digits or underscores

   Pattern: [a-zA-Z_][a-zA-Z0-9_]*

5. Write a program to check if the string is a palindrome and has even length. Examples: abba, noon, cc are valid. aba, racecar are not.

6. Accept strings where characters alternate between a and b, starting with a. Examples: abab, ab, ababab. Reject: aa, ba, abb.

Explore the codes in GitHub
Click or Scan the QR

Chapter 5

Finite Automata – NFA & DFA

Finite Automata are simple models used to process strings and check if they belong to a specific language. They are key in Lexical Analysis, helping compilers identify tokens like keywords, numbers, and identifiers in code.

In this chapter, we will first understand the formal definition of Finite Automata, represented using a five-tuple $(Q, \Sigma, \delta, q_0, F)$. We will then explore the two primary types of finite automata: Nondeterministic Finite Automata (NFA) and Deterministic Finite Automata (DFA), along with their graphical representations.

5.1. Finite Automata: A simple overview

A **Finite Automaton** (plural: **Finite Automata**) is a simple mathematical model used in computer science to check or recognize patterns in a series of symbols (like letters or numbers). It is mainly used in tasks like recognizing valid words, tokens, or identifiers in programming languages.

A finite automaton works by moving between a fixed number of **states** depending on the input it receives. Based on each symbol (like a letter or digit), it moves from one state to another. If the input follows the correct pattern, it ends in an **accepting state**; otherwise, it rejects the input.

Formal Definition of a Finite Automaton (FA)

A Finite Automaton is described as a **five-part (5-tuple)** model:

$$\text{FA} = (\mathbf{Q},\ \mathbf{\Sigma},\ \boldsymbol{\delta},\ \mathbf{q_0},\ \mathbf{F})$$

where,

- $Q \rightarrow$ A set of all possible states

- Σ (Sigma) $\rightarrow$ The set of input symbols (alphabet)

- δ (Delta) $\rightarrow$ The transition function (rules to move between states based on input)

- $q_0 \rightarrow$ The starting state ($q_0 \in Q$)

- $F \rightarrow$ A set of accepting/final states ($F \subseteq Q$)

Example: Consider a Finite Automaton that accepts strings ending in 'a'.
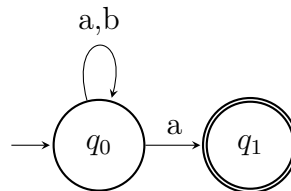
Figure 5.1: Finite automaton for strings ending in 'a'.

where:

- States $(Q) = \{q_0, q_1\}$

- Alphabet $(\Sigma) = \{a, b\}$

- Transition Function $(\delta) = q_0 \rightarrow q_0,\ q_0 \rightarrow q_1$

- Start State $(q_0) = q_0$

- Final States $(F) = \{q_1\}$

Types of Finite Automata:

There are mainly two types of finite automata:

1. Deterministic Finite Automaton (DFA)

2. Non-Deterministic Finite Automaton (NFA or NDFA)

5.2. Nondeterministic Finite Automaton

An NFA is like a magic maze where one input can open multiple doors at the same time! Unlike a normal automaton (DFA), which follows just one path, an NFA can explore many paths together for the same input.

It also has a special trick called ϵ **(epsilon) transitions**—these are like secret tunnels that let it move between states without reading any input.

This makes NFAs super flexible, but also harder to turn into real code (like in compilers) because they have to track all possible paths at once.

For Example: NFA that Accepts Strings Ending with "abb"
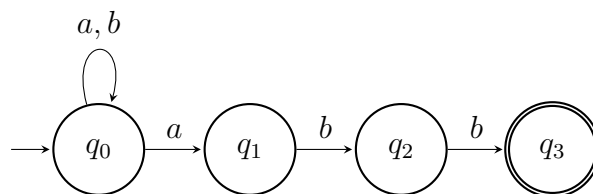
Figure 5.2: NFA accepting strings that end with "abb"

Consider the string: **aaaaabb**, i.e., five a's followed by two b's. The NFA remains in q_0 while reading the first four a's. When it reads the fifth a, it

jumps to q_1. Then it reads b, jumps to q_2, and finally to q_3.

The 5-tuple for this NFA is:

- **States** (Q): $\{q_0, q_1, q_2, q_3\}$

- **Alphabet** (Σ): $\{a, b\}$

- **Start state** ($q0$): q_0

- **Final states** (F): $\{q_3\}$

- **Transition function** (δ): A transition table is used for better understanding. Here $\emptyset$ is used if there is no transition of input symbol in a state.

State	Input a	Input b
q_0	$\{q_0, q_1\}$	$\{q_0\}$
q_1	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\{q_3\}$
q_3	$\emptyset$	$\emptyset$

Table 5.1: Transition Table

5.3. Deterministic Finite Automaton (DFA)

A Deterministic Finite Automaton (DFA) is a type of finite automaton where every input symbol results in exactly one defined state transition. Unlike a **Nondeterministic Finite Automaton (NFA)**, which allows multiple possible transitions for the same input, a DFA follows a single, unambiguous path for each symbol. This deterministic behavior makes DFAs more predictable and efficient, which is why they are widely used in compiler design and lexical analysis.

Another key feature of a DFA is that it does not allow ϵ-**transitions** (empty-string transitions). Every state change must be triggered by an input symbol,

ensuring that the automaton progresses clearly and systematically. This strict structure simplifies input processing, removes ambiguity, and avoids the need for backtracking during computation.

For Example: DFA for the language accepting strings ending with 'abba' over input alphabet: $\underline{\Sigma} = \{a, b\}$
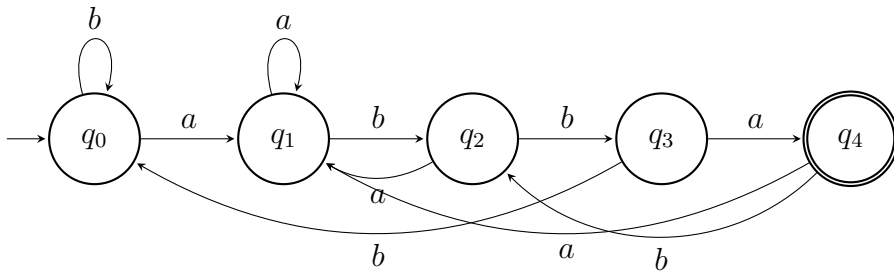
Figure 5.3: DFA accepting strings that end with "abba"

The 5-tuple for this DFA is:

- **States (Q):** $\{q_0, q_1, q_2, q_3, q_4\}$

- **Alphabet (Σ):** $\{a, b\}$

- **Start state(q0):** q_0

- **Final states (F):** $\{q_4\}$

- **Transition function (δ):** A transition table is used for better understanding. Here $\emptyset$ is used if there is no transition of input symbol in a state.

State	Input a	Input b
q_0	$\{q_1\}$	$\{q_0\}$
q_1	$\{q_1\}$	$\{q_2\}$
q_2	$\{q_1\}$	$\{q_3\}$
q_3	$\{q_4\}$	$\{q_0\}$
q_4	$\{q_1\}$	$\{q_2\}$

Table 5.2: Transition Table

5.4. Dead State in Finite Automata

In Finite Automata (FA), a **dead state** (also called a **trap state** or **sink state**) is a state from which no accepting (final) state can ever be reached. Once the automaton enters this state, it is stuck there and will never accept the input, no matter what symbols come next.

It's more common and important in DFA. They are used to explicitly handle invalid or unwanted input sequences. In practical applications, dead states are used to check the signal errors. For example, in lexical analysis, certain invalid sequences of characters can be directed to a dead state.

How to identify dead state:

A state q_d in a finite automaton is considered a dead state when, for every string w in the input alphabet Σ, the transition function δ leads q_d to itself or to another dead state.

Formally, $\delta(q_d, a) = q_d$ for every a in Σ.
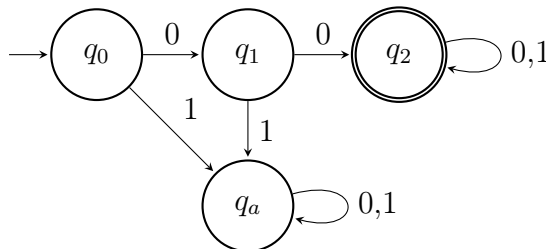
In this example, the state q_a can be considered as dead state. Here after we reach q_a by taking input 1 from q_0, it has no other place to go for inputs 0 and 1. In this diagram, q_0 is the initial state, q_2 is the accepting state (marked with double circle), q_a is the dead state - once entered, the automaton stays there for any input, and transitions to q_a represent invalid input sequences.

5.5. DFA vs NFA: Key Differences

Topic	DFA	NFA
1. Determinism	Single transition per input	Multiple transitions possible
2. ε-Transitions	Not allowed	Allowed
3. Path Exploration	Single path	Parallel paths
4. State Count	More states	Fewer states
5. Acceptance	Unique path to final state	Any path to final state

Table 5.3: Comparison between DFA and NFA

Extra Notes:

- In DFA, every state must have exactly one defined transition for each symbol in the input alphabet.

- In NFA, transitions can be missing, multiple, or ε-based.

- Every DFA can be represented as an NFA, and every NFA can be converted to an equivalent DFA, though the number of states may increase during conversion.

5.6. NFA to DFA Conversion

NFAs can have **multiple transitions** for the same input, or even ε-**transitions** (no input). Because of this, they are **not directly usable** in programs like compilers, which need clear, one-path logic.

So, we need to convert the NFA into a **DFA**, where each input from a state leads to only **one next state**. We do this using a method called the **Subset Construction Method**.

The Subset Construction Method

This method builds a DFA by grouping sets of NFA states into single DFA states. Steps to follow:

- Find all reachable NFA states for each input symbol from a state (including through ε-transitions if needed).

- Group those NFA states together as a single DFA state.

- Start state of DFA is the same as the NFA's start state (or the ε-closure of it).

- If any group (DFA state) contains a final state from the NFA, then mark it as a final state in the DFA too.

NFA to DFA Conversion Algorithm

Input: An NFA (with states, transitions, start state, final state).

Output: A DFA that works the same but in a deterministic way.

1. **Make a table for the NFA** — List all states, input symbols, and where each state goes for each input. If a state has multiple transitions for the same input, include all of them.

2. **Create an empty DFA table** — Make a blank table for the DFA with columns for each input symbol, just like the NFA. Leave the states empty for now.

3. **Start with the NFA's start state** — This will be the start state of the DFA. If the NFA has ε-transitions, include all states reachable through ε (ε-closure).

4. **Build new DFA states** — For each input symbol from a DFA state, find all NFA states that can be reached. Group them together to form a new DFA state.

5. **Repeat until done** — Keep checking all newly created DFA states. For each, repeat the process until no new DFA states are found.

6. **Mark the final states** — Any DFA state that contains one or more of the NFA's final states should be marked as a final state in the DFA.

Let us consider the NFA shown in the figure below:
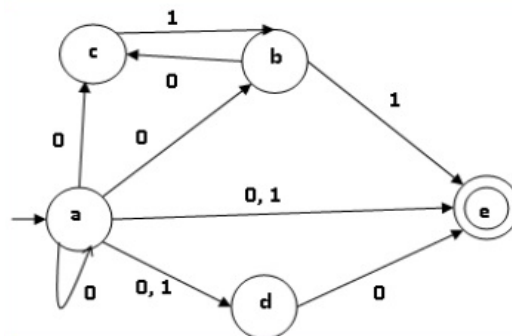
Figure 5.5: NFA Diagram

The transition table of this Figure 5.5 is given below, from where we will create a subset table starting with the initial state.

State	Input 0	Input 1
$\rightarrow a$	$\{a, b, c, d, e\}$	$\{d, e\}$
b	$\{c\}$	$\{e\}$
c	$\emptyset$	$\{b\}$
d	$\{e\}$	$\emptyset$
e	$\emptyset$	$\emptyset$

Table 5.4: Transition Table

Now a subset table will be made from the transition table following the Subset Construction algorithm.

DFA State	Input 0	Input 1
$\{a\}$	$\{a, b, c, d, e\}$	$\{d, e\}$
$\{a, b, c, d, e\}$	$\{a, b, c, d, e\}$	$\{b, d, e\}$
$\{d, e\}$	$\{e\}$	$\emptyset$
$\{b, d, e\}$	$\{c, e\}$	$\{e\}$
$\{e\}$	$\emptyset$	$\emptyset$
$\{c, e\}$	$\emptyset$	$\{b\}$
$\{b\}$	$\{c\}$	$\{e\}$
$\{c\}$	$\emptyset$	$\{b\}$

Table 5.5: Subset Table

From this subset table (Table 5.5), the DFA will be drawn maintaining the basic rules:
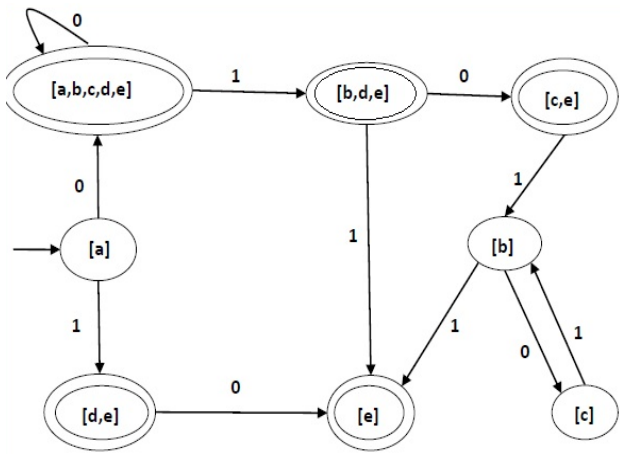
Figure 5.5: DFA Diagram

5.7. Problems and Exercises

5.7.1   Programming Problems

Problem 1: Write a program to check if a binary number is divisible by 3 using DFA.

```cpp
#include <iostream>
#include <string>
using namespace std;

int transition(int state, char input) {
    int bit = input - '0';
    return (state * 2 + bit) % 3;
}

bool isDivisibleBy3(string binary) {
    int state = 0;
    for (char c : binary) {
        if (c != '0' && c != '1') return false;
        state = transition(state, c);
    }
    return state == 0;
}

int main() {
    string input;
    cout << "Enter binary number: ";
    cin >> input;

    if (isDivisibleBy3(input))
        cout << "Accepted: Divisible by 3" << endl;
    else
        cout << "Rejected: Not divisible by 3" << endl;

    return 0;
}
```

Sample Input and Output

Input	Output
Enter binary number: 110	Accepted: Divisible by 3
Enter binary number: 101	Rejected: Not divisible by 3

Problem 2: NFA Simulation to Match Pattern `ab*`

```cpp
#include <iostream>
#include <string>
using namespace std;

bool simulateNFA(const string& input) {
    if (input.empty()) return false;
    if (input[0] != 'a') return false;

    for (size_t i = 1; i < input.length(); ++i) {
        if (input[i] != 'b') return false;
    }

    return true;
}

int main() {
    string input;
    cout << "Enter a string: ";
    cin >> input;

    if (simulateNFA(input))
        cout << "Accepted: Matches pattern ab*" << endl;
    else
        cout << "Rejected" << endl;

    return 0;
}
```

Sample Input and Output

Input	Output
Enter a string: a	Accepted: Matches pattern ab*
Enter a string: ab	Accepted: Matches pattern ab*
Enter a string: abbb	Accepted: Matches pattern ab*
Enter a string: b	Rejected
Enter a string: ba	Rejected

Problem 3: DFA Simulation to Accept Strings Ending with 01

```cpp
#include <iostream>
#include <string>
using namespace std;

bool endsWith01(const string& input) {
    int n = input.length();
    return (n >= 2 && input[n - 2] == '0' && input[n - 1] ==
        '1');
}

bool isBinary(const string& input) {
    for (char c : input) {
        if (c != '0' && c != '1') return false;
    }
    return true;
}

int main() {
    string input;
    cout << "Enter a binary string: ";
    cin >> input;

    if (!isBinary(input)) {
        cout << "Invalid input! Only 0 and 1 are allowed." <<
            endl;
    } else if (endsWith01(input)) {
        cout << "Accepted: Ends with 01" << endl;
    } else {
        cout << "Rejected" << endl;
    }

    return 0;
}
```

Sample Input and Output

Input	Output
Enter a binary string: 1101	Accepted: Ends with 01
Enter a binary string: 1010	Rejected
Enter a binary string: 01	Accepted: Ends with 01
Enter a binary string: 1111	Rejected
Enter a binary string: 10a1	Invalid input! Only 0 and 1 are allowed.

5.7.2   Exercises

Academic Exercises

1. Conceptual Questions

Q1. Define Finite Automata. Differentiate between DFA and NFA with an example.

Q2. Explain the formal definition of DFA. Describe each component briefly.

Q3. Can every NFA be converted to an equivalent DFA? Justify your answer.

Q4. Construct a DFA over {0,1} that accepts all strings ending with 01.

Q5. Draw the transition diagram of an NFA that accepts strings containing at least one 'a' and one 'b'.

Q6. Explain the difference between epsilon-NFA and standard NFA with examples.

Programming Exercises

1. DFA Programming Problems

P1. Write a C++ program that simulates a DFA which accepts binary numbers divisible by 3.

P2. Implement a DFA in C++ to accept strings ending in 01.

P3. Write a program that accepts binary strings with even number of 0s.

P4. Write a program that checks whether a string belongs to the language of a DFA defined by a given transition table.

2. NFA Programming Problems

P5. Simulate an NFA in C++ that accepts the language defined by the regular expression `ab*`.

P6. Write a program to simulate an epsilon-NFA that accepts strings ending with `a` or `aa`.

P7. Convert a given NFA transition table into DFA (you may use data structures to implement it).

Bonus: Practical Tips

- Use a state transition matrix or map to represent transitions.

- Store current states in a set while simulating NFAs.

- Test your programs with both valid and invalid strings.

- For visualization, use tools like JFLAP or write output to draw transitions.

Explore the codes in GitHub
Click or Scan the QR

Chapter 6

Left Recursion & Left Factoring

6.1. Left Recursion

A grammar is **left recursive** if a non-terminal symbol has a production rule where it recurses on itself as the leftmost symbol. That is, $A \rightarrow A\alpha$ for some string α. This form of recursion causes problems in top-down parsers because it can lead to infinite loops during parsing. Left recursion must be eliminated to make a grammar compatible with recursive descent parsers.

6.1.1 Immediate Left Recursion

Immediate left recursion occurs when a non-terminal directly refers to itself on the left side of a production. Direct self-reference in production:

$$A \rightarrow A\alpha \mid \beta$$

It is immediately left-recursive. This is the simplest type of left recursion and can be directly transformed into a right-recursive form using helper rules.

6.1.2 Indirect Left Recursion

In indirect left recursion, the recursion occurs through multiple intermediate non-terminals. Chain of productions leading back to original non-terminal:

$$A \rightarrow B\alpha, \quad B \rightarrow A\beta$$

To resolve indirect recursion, we must reorder the grammar and then apply immediate recursion elimination techniques.

6.1.3   Problems Caused by Left Recursion

Left recursion is problematic for top-down parsers, especially LL(1), because it causes infinite recursion. As the parser keeps calling the same function without consuming input tokens, it may never terminate. Removing left recursion is essential to ensure the grammar can be parsed efficiently and deterministically.

- Infinite loops in top-down parsers

- LL(1) parser conflicts

- Non-termination in recursive descent

6.1.4   Immediate Left Recursion Removal

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

to:
$$A \rightarrow \beta A'$$
$$A' \rightarrow \alpha A' \mid \varepsilon$$

Example 1:

Original:
$$E \rightarrow E + T \mid T$$

Eliminated:
$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \varepsilon$$

Example 2:

Original:
$$A \rightarrow AB \mid a$$

Eliminated:
$$A \rightarrow aA'$$
$$A' \rightarrow BA' \mid \varepsilon$$

Example 3:

Original:
$$S \rightarrow S(S) \mid \varepsilon$$

Eliminated:
$$S \rightarrow \varepsilon S'$$
$$S' \rightarrow (S)S' \mid \varepsilon$$

Example 4:

Original:
$$F \rightarrow F * n \mid n$$

Eliminated:
$$F \rightarrow nF'$$
$$F' \rightarrow *nF' \mid \varepsilon$$

Example 5:

Original:
$$T \rightarrow T/F \mid F$$

Eliminated:
$$T \rightarrow FT'$$
$$T' \rightarrow /FT' \mid \varepsilon$$

6.1.5   Indirect Left Recursion Removal

Example 6:

Original:
$$A \rightarrow Bb \mid a$$
$$B \rightarrow Aa \mid c$$

Eliminated:
$$A \rightarrow aA' \mid cbA'$$
$$A' \rightarrow abA' \mid \varepsilon$$
$$B \rightarrow Aa \mid c$$

Example 7:

Original:
$$X \rightarrow Yz \mid x$$
$$Y \rightarrow Xy \mid w$$

Eliminated:
$$X \rightarrow xX' \mid wzX'$$
$$X' \rightarrow yzX' \mid \varepsilon$$
$$Y \rightarrow Xy \mid w$$

Example 8:

Original:
$$P \rightarrow Qx \mid p$$
$$Q \rightarrow Py \mid q$$

Eliminated:
$$P \rightarrow pP' \mid qxP'$$
$$P' \rightarrow yxP' \mid \varepsilon$$
$$Q \rightarrow Py \mid q$$

Example 9:

Original:
$$M \to NOP \mid NOQ \mid R$$
$$N \to MN \mid \varepsilon$$

Eliminated:
$$M \to RM' \mid NOPM' \mid NOQM'$$
$$M' \to NM' \mid \varepsilon$$
$$N \to MN \mid \varepsilon$$

Example 10:

Original:
$$C \to Dd \mid e$$
$$D \to Cf \mid g$$

Eliminated:
$$C \to eC' \mid gdC'$$
$$C' \to fdC' \mid \varepsilon$$
$$D \to Cf \mid g$$

6.2. Left Factoring

Left factoring is a technique used to make a grammar suitable for top-down parsing when two or more productions for a non-terminal begin with the same prefix. By factoring out the common part and introducing a new non-terminal, we make the grammar deterministic and easier to parse using LL(1) parsers.

Formally, for productions:

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

where α is the common prefix.

6.2.1   Why Left Factoring is Needed

If a parser cannot determine which production to use based on the first symbol, ambiguity arises. Left factoring resolves this by isolating the common part, allowing the parser to decide based on the next input token. This helps prevent parsing errors and improves predictability.

- Resolves **FIRST-FIRST conflicts** in LL(1) parsers

- Eliminates **backtracking** in top-down parsers

- Makes grammar **deterministic** for predictive parsing

- Required for **LL(k) parser** implementation

- Reduces **ambiguity** in grammar rules

6.2.2   Left Factoring Algorithm

1. Identify common prefixes in the right-hand sides of a production.

2. Factor out the common prefix.

3. Introduce a new non-terminal to handle the remaining options.

4. Replace the original production with the factored version.

5. Repeat the process until no common prefixes remain.

Given productions with common prefix α:

- Group all productions:

  $A \rightarrow \alpha\beta_1 \,|\, \alpha\beta_2 \,|\, \cdots \,|\, \alpha\beta_n \,|\, \gamma$

- Create new non-terminal: A'

- Rewrite as:

$$A \to \alpha A' \mid \gamma$$
$$A' \to \beta_1 \mid \beta_2 \mid \cdots \mid \beta_n$$

6.2.3   Examples of Left Factoring

Example 1:

Original:
$$S \to aB \mid aC$$

Factored:
$$S \to aS'$$
$$S' \to B \mid C$$

Example 2:

Original:
$$A \to abC \mid abD \mid c$$

Factored:
$$A \to abA' \mid c$$
$$A' \to C \mid D$$

Example 3:

Original:

$$\text{stmt} \to \text{if expr then stmt else stmt} \mid \text{if expr then stmt}$$

Factored:
$$\text{stmt} \to \text{if expr then stmt stmt}'$$
$$\text{stmt}' \to \text{else stmt} \mid \varepsilon$$

Example 4:

Original:
$$E \to T + E \mid T - E \mid T$$

Factored:
$$E \to TE'$$
$$E' \to +E \mid -E \mid \varepsilon$$

Example 5:

Original:
$$F \to \text{num} * \text{num} \mid \text{num}/\text{num} \mid \text{num}$$

Factored:
$$F \to \text{num } F'$$
$$F' \to *\text{num} \mid /\text{num} \mid \varepsilon$$

Example 6:

Original:
$$X \to Yz \mid Yw \mid a$$

Factored:
$$X \to YX' \mid a$$
$$X' \to z \mid w$$

Example 7: **Original:**
$$P \to Qx \mid Qy \mid R$$

Factored:
$$P \to QP' \mid R$$
$$P' \to x \mid y$$

Example 8:

Original:
$$A \rightarrow Bcd \,|\, Bce \,|\, f$$

Factored:
$$A \rightarrow BA' \,|\, f$$
$$A' \rightarrow cd \,|\, ce$$

Example 9:

Original:
$$C \rightarrow D + E \,|\, D - E \,|\, g$$

Factored:
$$C \rightarrow DC' \,|\, g$$
$$C' \rightarrow +E \,|\, -E$$

Example 10:

Original:
$$Z \rightarrow XY \,|\, XZ \,|\, a$$

Factored:
$$Z \rightarrow XZ' \,|\, a$$
$$Z' \rightarrow Y \,|\, Z$$

Example 11:

Original:
$$K \rightarrow Lm \,|\, Ln \,|\, o$$

Factored:
$$K \rightarrow LK' \,|\, o$$
$$K' \rightarrow m \,|\, n$$

Example 12:

Original:
$$G \to HIJ \,|\, HIK \,|\, L$$

Factored:
$$G \to HIG' \,|\, L$$
$$G' \to J \,|\, K$$

Example 13:

Original:
$$W \to UVX \,|\, UVY \,|\, Z$$

Factored:
$$W \to UVW' \,|\, Z$$
$$W' \to X \,|\, Y$$

Example 14:

Original:
$$T \to Sa \,|\, Sb \,|\, c$$

Factored:
$$T \to ST' \,|\, c$$
$$T' \to a \,|\, b$$

Example 15:

Original:
$$M \to NOP \,|\, NOQ \,|\, R$$

Factored:
$$M \to NOM' \,|\, R$$
$$M' \to P \,|\, Q$$

6.3. Problems and Exercises

6.3.1   Programming Problems

Problem 1: Write a C program that takes a grammar production rule as input(e.g., A → Aα | β) and detects if it has left recursion.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char lhs[10], prod[100];
    printf("Enter LHS of production: ");
    scanf("%s", lhs);
    printf("Enter RHS productions (use | to separate): ");
    scanf("%s", prod);

    char *token = strtok(prod, "|");
    int found = 0;

    while (token != NULL) {
        if (strncmp(token, lhs, strlen(lhs)) == 0) {
            found = 1;
            break;
        }
        token = strtok(NULL, "|");
    }

    if (found)
        printf("Left Recursion Detected\n");
    else
        printf("No Left Recursion\n");

    return 0;
}
```

Sample Input and Output

Input	Output
LHS: A	Left Recursion Detected
RHS: Aa|b	
LHS: A	No Left Recursion
RHS: bA|c	

Problem 2: Write a C program that removes immediate left recursion from a rule like $A \rightarrow A\alpha \mid \beta$ and rewrites it to $A \rightarrow \beta A'$, $A' \rightarrow \alpha A' \mid \epsilon$

```c
#include <stdio.h>
#include <string.h>

int main() {
    char lhs[10], prod[100], alpha[10][20], beta[10][20];
    int ac = 0, bc = 0;

    printf("Enter LHS: ");
    scanf("%s", lhs);
    printf("Enter RHS (use | to separate): ");
    scanf("%s", prod);

    char *token = strtok(prod, "|");
    while (token != NULL) {
        if (strncmp(token, lhs, strlen(lhs)) == 0)
            strcpy(alpha[ac++], token + strlen(lhs));
        else
            strcpy(beta[bc++], token);
        token = strtok(NULL, "|");
    }

    if (ac == 0)
        printf("No Left Recursion\n");
    else {
        printf("%s -> ", lhs);
        for (int i = 0; i < bc; i++)
            printf("%s%s' %s", beta[i], lhs, (i < bc - 1) ? "
                | " : "\n");

        printf("%s' -> ", lhs);
        for (int i = 0; i < ac; i++)
            printf("%s%s' %s", alpha[i], lhs, (i < ac - 1) ?
                "| " : "| \n");
    }

    return 0;
}
```

Sample Input and Output

Input	Output
LHS: A	A → bA'
RHS: Aa|b	A' → aA' |

Problem 3: Write a C program that finds the longest common prefix from multiple productions.

```c
#include <stdio.h>
#include <string.h>

int main() {
    char prod[3][100];
    printf("Enter 3 productions:\n");
    for (int i = 0; i < 3; i++)
        scanf("%s", prod[i]);

    int minLen = strlen(prod[0]);
    for (int i = 1; i < 3; i++) {
        if (strlen(prod[i]) < minLen)
            minLen = strlen(prod[i]);
    }

    int i;
    for (i = 0; i < minLen; i++) {
        if (prod[0][i] != prod[1][i] || prod[0][i] != prod
            [2][i])
            break;
    }

    if (i > 0) {
        printf("Common Prefix: ");
        for (int j = 0; j < i; j++)
            printf("%c", prod[0][j]);
        printf("\n");
    } else {
        printf("No Common Prefix\n");
    }

    return 0;
}
```

Sample Input and Output

Input	Output
abcd	Common Prefix: ab
abxy	
abpq	
cat	No Common Prefix
dog	
man	

Problem 4:   Given a rule like A → abc | abd, convert it to A → abX, X → c | d

```
#include <stdio.h>
#include <string.h>

int main() {
    char p1[100], p2[100];
    printf("Enter production 1: ");
    scanf("%s", p1);
    printf("Enter production 2: ");
    scanf("%s", p2);

    int i = 0;
    while (p1[i] == p2[i]) i++;

    printf("Factored Production:\n");
    printf("A -> ");
    for (int j = 0; j < i; j++)
        printf("%c", p1[j]);
    printf("X\n");

    printf("X -> ");
    printf("%s | %s\n", p1 + i, p2 + i);

    return 0;
}
```

Sample Input and Output

Input	Output
abc	A → abX
abd	X → c | d

6.3.2 Exercises

A. Multiple Choice Questions (MCQ)

1. Which of the following productions is left recursive?
   (a) A → A a | b
   (b) A → b A | a
   (c) A → a A | a
   (d) A → a b | b a

2. What is the purpose of eliminating left recursion?
   (a) To simplify grammar
   (b) To convert grammar into LL(1) form
   (c) To improve parse tree height
   (d) To convert ambiguous grammar

3. Which of the following is a correct transformation of left recursion?
   A → Aα | β becomes:
   (a) A → βA′, A′ → αA′ | ε
   (b) A → αβ
   (c) A → Aβ | α
   (d) A → β | α

4. Left factoring is used to:
   (a) Detect ambiguity in grammar
   (b) Remove redundancy in terminals
   (c) Factor out common prefixes
   (d) Convert grammar into regular expression

5. Which one causes difficulty in recursive descent parsing?
   (a) Left recursion
   (b) Right recursion
   (c) Left factoring
   (d) None

B. Short Questions

1. Define immediate left recursion and indirect left recursion with examples.

2. What is the difference between left recursion and right recursion?

3. What is left factoring? Why is it used in parser design?

4. How does left recursion affect LL(1) parsers?

5. What are the steps to eliminate immediate left recursion from a production?

C. Descriptive Questions

1. Eliminate left recursion from the following grammar: `A → Aa | Ab | c`

2. Perform left factoring on the following production: `S → if E then S else S | if E then S`

3. Explain with example how to remove both immediate and indirect left recursion.

4. Consider the grammar: `E → E+T | T  T → T*F | F  F → (E) | id` Remove left recursion and rewrite it in LL(1) form.

5. Write an algorithm or program to perform left factoring for a set of grammar rules.

D. Programming Problems (from 6.3.1)

1. Write a C program to detect immediate left recursion in a given production.

2. Write a program to eliminate immediate left recursion from a grammar rule.

3. Write a program that finds and factors out the common prefix in mul-

tiple productions.

4. Write a program that performs left factoring on two given alternatives.

Explore the codes in GitHub
Click or Scan the QR

Chapter 7

First, Follow and LL(1) Parsing Table

Parsing is an important step in understanding the structure of a program by checking if it follows the correct grammar rules. To make parsing easier and faster, we use two helpful sets called **First** and **Follow**.

The **First** set of a non-terminal symbol tells us which terminals (actual characters or tokens) can appear at the very beginning of strings that this non-terminal can produce. On the other hand, the **Follow** set tells us which terminals can come right after that non-terminal in a valid sentence.

These sets are very useful when building a predictive parser, like an LL(1) parser, because they help decide which grammar rule to apply next without needing to try multiple possibilities or backtrack. In this chapter, we will learn what First and Follow sets are, how to find them, and how to use them to create a parsing table for LL(1) parsing.

7.1. Guide to the First Set

The First set of a non-terminal symbol A is the collection of terminals that can appear at the very beginning of any string that A can produce. This helps the parser decide which rule to use when it tries to replace A during parsing.

Rules to Compute First Set

1. If X is a **terminal** (like a letter or number), then $\text{First}(X)$ is just $\{X\}$ itself.

2. If X is a **non-terminal** and has a rule $X \rightarrow Y_1 Y_2 \ldots Y_n$, then:

   - First, look at Y_1 and add its First set to $\text{First}(X)$ (but don't add ε here).

   - If Y_1 can be empty (ε), then check Y_2 and add its First set too.

   - Keep doing this for each Y_i until you find one that cannot be empty.

   - If all Y_1 to Y_n can be empty, then add ε to $\text{First}(X)$.

3. If X can directly produce ε (empty string), add ε to $\text{First}(X)$.

For example, here is a grammar:

$$X \rightarrow YZ \mid d$$
$$Y \rightarrow e \mid a$$
$$Z \rightarrow f \mid g \mid a$$

The First sets for each non-terminal are:

Non-terminal	First()
X	$\{d, e, f, g, a\}$
Y	$\{e, a\}$
Z	$\{f, g, a\}$

In case of finding the first set, it is better to start from the bottom, this will help finding the set faster and in an easier way.

7.2. Guide to the Follow Set

The Follow set of a non-terminal tells us what terminals (symbols) can appear immediately after that non-terminal in some valid sentence. This is very important in LL(1) predictive parsing, where the parser uses the Follow set to know when a rule has finished and what to expect next.

Rules to Compute Follow Set

- **Start Symbol Rule**: Add the end-of-input marker **$** to the Follow set of the start symbol.

- **If A $\rightarrow \alpha$ Bβ**: Add everything from First(β) (except ε) to Follow(B), since β might come right after B.

- **If A $\rightarrow \alpha$ B**: If B is at the end of a production, then add Follow(A) to Follow(B)

- **If A $\rightarrow \alpha$ Bβ, and First(β) contains** ε: Add both First(β) - $\{\varepsilon\}$ and Follow(A) to Follow(B), because sometimes β might not appear at all.

For example, here is a grammar:

$$S \rightarrow AB$$
$$A \rightarrow a \mid \varepsilon$$
$$B \rightarrow b \mid c$$

The Follow sets for each non-terminal are:

Non-terminal	Follow()
S	$\{\$ \}$
A	$\{b, c\}$
B	$\{\$ \}$

When computing Follow sets, first construct the First set and then determine the Follow set based on the grammar rules.

7.3. Example: Find First & Follow Set

(Try to solve it yourself first, then compare your result with the solution)

Consider This Grammar:

$$S \rightarrow ACB \mid CbB \mid Ba$$
$$A \rightarrow da \mid Bc$$
$$B \rightarrow g \mid \varepsilon$$
$$C \rightarrow h \mid \varepsilon$$

The First and Follow Sets are:

Non-terminal	First()	Follow()
S	$\{d, g, h, b, a\}$	$\{\$\}$
A	$\{d, g\}$	$\{h, g, \$\}$
B	$\{g, \varepsilon\}$	$\{\$, a, c\}$
C	$\{h, \varepsilon\}$	$\{g, \$, b\}$

7.4. LL(1) Parsing and Parsing Table

LL(1) parsing is a top-down parsing technique. Here's what LL(1) means:

- **L** – Read the input from **Left** to right

- **L** – Produce a **Leftmost** derivation

- **1** – Use **1** lookahead symbol to decide which rule to apply

This kind of parser does not use backtracking. It uses a special table (called

the parsing table) to decide what to do next, based on the current input symbol and the top of the stack.

Steps to Construct an LL(1) Parsing Table

- For each production $A \rightarrow \alpha$, place it in Table$[A, X]$ for every $X \in$ First(α) (excluding ε).

- If $\varepsilon \in$ First(α), then also place $A \rightarrow \alpha$ in Table$[A, Y]$ for every $Y \in$ Follow(A).

- If any table cell has more than one entry, it causes a conflict, meaning the grammar is not LL(1).

- **Example 1**: Consider the given grammar:

$$S \rightarrow A\ B$$
$$A \rightarrow a \mid \varepsilon$$
$$B \rightarrow b\ C$$
$$C \rightarrow c \mid \varepsilon$$

The First & Follow sets for the above grammar are:

Non-terminal	First()	Follow()
S	$\{a, b\}$	$\{\$\}$
A	$\{a, \varepsilon\}$	$\{b\}$
B	$\{b\}$	$\{\$\}$
C	$\{c, \varepsilon\}$	$\{\$\}$

Now based on the first and follow sets the LL(1) parsing table is constructed below:

Non-terminal	a	b	c	$
S	$S \to A\ B$	$S \to A\ B$		
A	$A \to a$	$A \to \varepsilon$		
B		$B \to b\ C$		
C			$C \to c$	$C \to \varepsilon$

Each cell has only one production, no conflicts. So, **Grammar is LL(1).**

- **Example 2:** Consider the below Grammar:

$$S \to A\ B$$
$$A \to a \mid \varepsilon \mid c$$
$$B \to b \mid A\ c$$

The first and follow set for this grammar:

Non-terminal	First()	Follow()
S	$\{a, b, c\}$	$\{\$\}$
A	$\{a, \varepsilon, c\}$	$\{a, b, c\}$
B	$\{a, b, c\}$	$\{\$\}$

Now based on the first and follow sets the LL(1) parsing table is constructed below:

Non-terminal	a	b	c	$
S	$S \to A\ B$	$S \to A\ B$	$S \to A\ B$	
A	$A \to a\ /\ A \to \varepsilon$	$A \to \varepsilon$	$A \to \varepsilon$	
B	$B \to A\ c$	$B \to b$	$B \to A\ c$	

Table[A, a] contains two productions ($A \to a$ / $A \to \varepsilon$). This means the parser cannot decide which rule to use on input a. So, **this grammar is not LL(1).**

7.5. Problems and Exercises

7.5.1   Programming Problems

Problem 1: Given a grammar rule like A → aB | ε, write a C program that reads the production(s) and outputs the FIRST set for non-terminal A.

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>

int main() {
    char prod[10][20];
    int n;
    char nt;

    printf("Enter number of productions: ");
    scanf("%d", &n);
    printf("Enter productions:");

    for (int i = 0; i < n; i++) {
        scanf("%s", prod[i]);
    }

    nt = prod[0][0]; // First production's LHS

    printf("FIRST(%c) = { ", nt);
    int firstPrinted = 0;

    for (int i = 0; i < n; i++) {
        char symbol = prod[i][3]; // First symbol after A->

        if (islower(symbol) || symbol == 'e') { // use 'e'
           for epsilon
            if (firstPrinted) printf(", ");
            printf("%c", symbol);
            firstPrinted = 1;
        }
        else if (isupper(symbol)) {
            if (firstPrinted) printf(", ");
```

```
                printf("[Need recursion for %c]", symbol);
                firstPrinted = 1;
            }
        }
        printf(" }\n");

        return 0;
}
```

Sample Input and Output

Input	Output
2	FIRST(A) = { a, ε }
A -> aB	
A -> ε	

Problem 2: Write a C program to compute and print the First sets for each non-terminal S, A, and B. The given grammar(S -> A B , A -> a | b , B -> c | d)

```c
#include <stdio.h>
#include <string.h>

#define MAX 10

// We only handle this specific grammar: S->AB, A->a|b, B->c|
    d

void computeFirst(char first[][MAX]) {
    // First(A) = {a, b}
    strcpy(first[0], "ab");   // index 0 = A

    // First(B) = {c, d}
    strcpy(first[1], "cd");   // index 1 = B

    // First(S) = First(A)
    strcpy(first[2], first[0]);   // index 2 = S
}

void printFirst(char first[][MAX]) {
    printf("First sets:\n");
    printf("First(A) = { ");
    for (int i = 0; first[0][i] != '\0'; i++)
        printf("%c ", first[0][i]);
    printf("}\n");

    printf("First(B) = { ");
    for (int i = 0; first[1][i] != '\0'; i++)
        printf("%c ", first[1][i]);
    printf("}\n");

    printf("First(S) = { ");
    for (int i = 0; first[2][i] != '\0'; i++)
        printf("%c ", first[2][i]);
    printf("}\n");
}
```

```
int main() {
    char first[3][MAX];   // 0 = A, 1 = B, 2 = S

    computeFirst(first);

    printFirst(first);

    return 0;
}
```

Sample Input and Output

Input	Output
Grammar:	First sets:
$S \rightarrow AB$	First(A) = { a, b }
$A \rightarrow a \mid b$	First(B) = { c, d }
$B \rightarrow c \mid d$	First(S) = { a, b }

7.5.2 Exercises

1. Compute FIRST and FOLLOW sets for Given grammar:

$$E \rightarrow TE'$$
$$E' \rightarrow +TE' \mid \epsilon$$
$$T \rightarrow FT'$$
$$T' \rightarrow *FT' \mid \epsilon$$
$$F \rightarrow (E) \mid \text{id}$$

2. Consider the grammar

$$S \rightarrow AB$$
$$A \rightarrow aA \mid \epsilon$$
$$B \rightarrow bB \mid \epsilon$$

- Find FIRST and FOLLOW for all non-terminals

- Is this grammar suitable for <u>LL(1)</u> parsing?

3. Compute FIRST and FOLLOW sets for all non-terminals.

$$S \rightarrow aA \mid bB$$
$$A \rightarrow a \mid S$$
$$B \rightarrow b \mid S$$

4. Consider this grammar:

$$E \rightarrow E + T \mid T$$
$$T \rightarrow T * F \mid F$$
$$F \rightarrow [E] \mid \text{id}$$

- Identify why this grammar is not LL(1)

Explore the codes in GitHub
Click or Scan the QR

Chapter 8

LR(0) Parser and Canonical Table

Parsing is an important part of compiler design. It helps us check whether a given input follows the rules of a programming language's grammar. There are different parsing methods, and one of them is LR(0) parsing. Unlike LL(1) parsing, which works from the top down, LR(0) parsing works from the bottom up. This approach can handle more complex grammars. LR(0) parsers use a method called shift-reduce parsing, where input symbols are either shifted onto a stack or reduced using grammar rules. To make decisions during parsing, LR(0) uses something called a canonical collection of LR(0) items. These items help build a parsing table, which guides the parser step by step.

In this chapter, we'll learn about LR(0) parsing, its rules, and how to build the parsing table using these items.

8.1. LR(0) Items

An LR(0) item is a production rule with a dot $(\cdot)$ indicating how much of the input has been seen so far during parsing. If a production is $A \rightarrow XYZ$, then the LR(0) items derived from this would be:

- $A \rightarrow \cdot XYZ$ (nothing is parsed yet)

- $A \rightarrow X \cdot YZ$ (X is parsed)

- $A \rightarrow XY \cdot Z$ (X and Y are parsed)

- $A \rightarrow XYZ\cdot$ (entire right-hand side is parsed)

The dot helps track parser progress through the production during parsing.

8.2. Augmented Grammar

Before building LR(0) items, we first augment the grammar: This means we add a new start symbol, say S', and a new production rule:

- $S' \rightarrow S$

This helps define a unique accept state in the parser. It clarifies when parsing is complete: when we see the item $S' \rightarrow S\cdot$.

8.3. Closure and GOTO Functions

Closure Function

Given a set of LR(0) items, the closure function adds more items based on what's after the dot.

Steps:

1. Start with a set of LR(0) items (usually just one).

2. For each item like $A \rightarrow \alpha \cdot B\beta$, where B is a non-terminal, add all productions of B with the dot at the beginning: $B \rightarrow \cdot\gamma$

3. Repeat until no more items can be added.

GOTO Function

The $GOTO(I, X)$ function defines the next item set when the parser sees a symbol X from state I.

Steps:

1. For each item in I of form $A \rightarrow \alpha \cdot X\beta$, move the dot past X: $A \rightarrow \alpha X \cdot \beta$

2. Then apply closure to the resulting items.

8.4.   LR(0) Parsing and Its Basic Rules

LR(0) parsing is a bottom-up technique used in compilers to analyze input using shift-reduce parsing, and it does not use any lookahead symbol. It builds a set of items (called LR(0) items) to decide how to parse the input step by step.

Here are the main rules it follows:

1. **Add Augmented Production**
   First, we add a new starting production like $S' \rightarrow \bullet S$, where S is the original start symbol. The dot ($\bullet$) shows how much of the rule we have read so far.

2. **Compute Closure for I_0 (Initial State)**
   Start with the new production and find all related productions where the dot is at the beginning. This set is called the closure of state I_0.

3. **Apply GOTO for Each Symbol**
   For every state and symbol (like terminals or non-terminals), move the dot one position to the right and create a new state. This is done using the GOTO function.

4. **Repeat Until No New States**
   Keep applying the closure and GOTO steps for new states until no more new states are created. This gives us the canonical collection of

LR(0) items.

5. **Build the Parsing Table**
   Using all the states created, we now make the LR(0) parsing table, which has two parts:

   - **ACTION table** (for terminals): shift, reduce, accept, or error

   - **GOTO table** (for non-terminals): shows which state to move to

   This full process helps the parser work correctly and efficiently.

8.5. LR(0) Parsing Actions

- Shift (S) $\rightarrow$ Move the next input symbol to the stack and go to the next state.

- Reduce (r) $\rightarrow$ Replace part of the stack using a production rule (right side $\rightarrow$ left side).

- Accept $\rightarrow$ If all input is successfully parsed and the start symbol is reached, parsing is complete.

- Error $\rightarrow$ If no rule matches, the input is invalid.

8.6. Canonical Collection & Table of LR(0)

To create an LR(0) parsing table, we first need to generate a canonical collection of LR(0) items. This is done using two main functions: closure and goto.

- **Shift Move**
  If a state goes to another state on a terminal symbol (like a, b, $+$, $*$, etc.), it means a shift action will be added to the table.

- **Goto Move**
  If a state goes to another state on a non-terminal (variable) (like E, T, S), it will be added in the GOTO part of the table.

- **Reduce Move**
  If a state has a final item (a rule where the dot is at the end, like $A \rightarrow \alpha\bullet$), then we add a reduce action in the table for that rule.

8.6.1 Example of LR(0) Parsing

Consider the grammar:

$$E \rightarrow BB$$
$$B \rightarrow cB \mid d$$

The augmented grammar with production numbers:

$$E' \rightarrow E \quad 0$$
$$E \rightarrow BB \quad 1$$
$$B \rightarrow cB \quad 2$$
$$B \rightarrow d \quad 3$$

Then the LR(0) Parsing is:
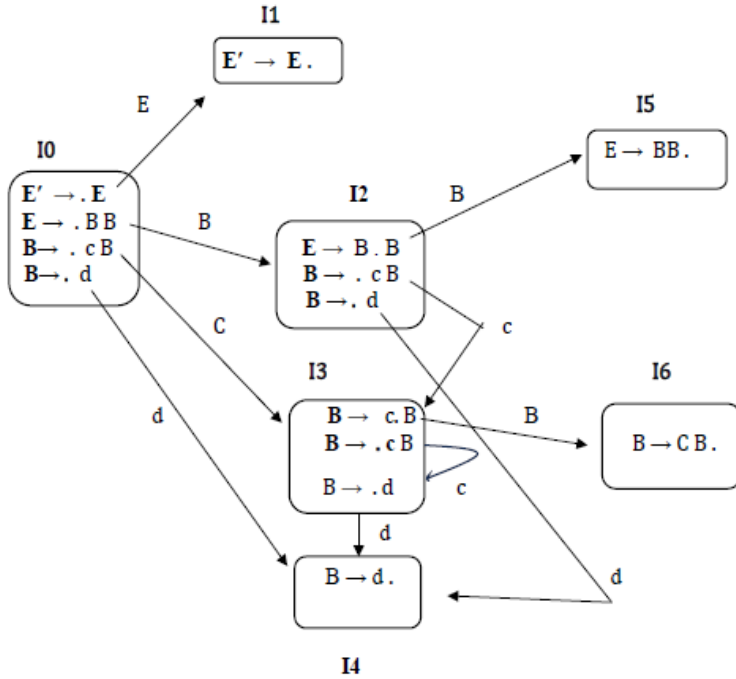
Figure 8.1: Canonical LR(0) Parsing States

Here,

I_0: Initial state with augmented production and closure applied.

I_1: GOTO(I_0, E) $\rightarrow$ Accept state. Contains item $E' \rightarrow E\cdot$

I_2: GOTO(I_0, B) $\rightarrow$ Expand with productions of B. Dot before B in $E \rightarrow B \cdot B$

I_3: GOTO(I_2, c) and GOTO(I_0, c) $\rightarrow$ Expand with production of B.

I_4: GOTO(I_3, d) and GOTO(I_0, d) $\rightarrow$ Closure on $B \rightarrow d$.

I_5: GOTO(I_2, B) $\rightarrow$ Shift dot in $E \rightarrow BB\cdot$

I_6: GOTO(I_3, B) → Shift dot in $B \rightarrow cB\cdot$

Canonical Table:

State	Action			GOTO	
	c	**d**	**$**	**E**	**B**
I0	S3	S4		1	2
I1			accepted		
I2	S3	S4			5
I3	S3	S4			6
I4	r3	r3	r3		
I5	r1	r1	r1		
I6	r2	r2	r2		

Explanation:

- I_0 on S goes to I_1, so GOTO(S) = 1.

- I_0 on A goes to I_2, so GOTO(A) = 2.

- I_2 on A goes to I_5, so GOTO(A) = 5.

- I_3 on A goes to I_6, so GOTO(A) = 6.

- I_0, I_2, and I_3 on a go to I_3, so ACTION = S3 (shift 3).

- I_0, I_2, and I_3 on b go to I_4, so ACTION = S4 (shift 4).

- I_4 has $A \rightarrow a\cdot$, so it's a final item ⇒ reduce by production 3 (r3).

- I_5 has $S \rightarrow A\cdot$, so it's a final item ⇒ reduce by production 1 (r1).

- I_6 has $A \rightarrow b\cdot$, so it's a final item ⇒ reduce by production 2 (r2).

- I_1 has $S' \rightarrow S\cdot$ with end marker $, so ACTION = accept.

8.7. Problems and Exercises

8.7.1 Programming Problems

Problem 1: Check if a grammar is LR(0) compatible

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;

// Placeholder - Full LR(0) item set generation is complex
// This sample checks for presence of a specific conflict
   manually

int main() {
    vector<string> productions = {
        "S->A",
        "A->aA",
        "A->b"
    };

    // Naively assume no conflicts
    cout << "Productions:" << endl;
    for (string prod : productions)
        cout << prod << endl;

    cout << "\nNo shift/reduce conflict found. Grammar is LR
        (0) compatible.\n";
    return 0;
}
```

Sample Input and Output

Input	Output
(Sample production list hard-coded)	Grammar is LR(0) compatible.

Problem 2: Simulate a simple LR(0) shift-reduce parser

Implement a shift-reduce parser for the grammar:

$$E \rightarrow E + T \mid T, \quad T \rightarrow T * F \mid F, \quad F \rightarrow (E) \mid id$$

```cpp
#include <iostream>
#include <stack>
#include <string>
using namespace std;

// This is a toy simulation, assumes token stream is like "id
    +id*id"

bool isOperator(char c) {
    return c == '+' || c == '*';
}

int main() {
    string input = "id+id";
    stack<string> st;
    int i = 0;

    cout << "Input: " << input << endl;
    while (i < input.length()) {
        if (input[i] == 'i' && input[i+1] == 'd') {
            st.push("F");
            i += 2;
        } else if (isOperator(input[i])) {
            st.push(string(1, input[i]));
            i++;
        } else {
            cout << "Invalid token\n";
            return 0;
        }
    }

    cout << "Final Stack Top: " << st.top() << endl;
    cout << "Simulated parsing complete.\n";
    return 0;
}
```

Input	Output
`id+id`	Final Stack Top: F
Simulated parsing complete.	

Problem 3: Write a program to generate the canonical collection of LR(0) items for a grammar like:

$$S' \rightarrow S, \quad S \rightarrow aSa \mid bSb \mid \varepsilon$$

```cpp
#include <iostream>
#include <vector>
#include <string>
using namespace std;

// Only generates initial item [S' -> .S]
int main() {
    string start = "S'";
    string production = "S";

    cout << "Initial LR(0) item:\n";
    cout << start << " -> ." << production << endl;

    // In real parser, closure and goto functions are needed
    cout << "Further states need full closure and goto
        simulation.\n";
    return 0;
}
```

Sample Input and Output

Input	Output
(Production hardcoded)	S' → .S
Further states need full closure	
and goto simulation.	

8.7.2 Exercises

Academic Exercises

1. Construct the LR(0) items for the grammar:

$$S \to E$$
$$E \to E + T \,|\, T$$
$$T \to T * F \,|\, F$$
$$F \to (E) \,|\, id$$

2. For the above grammar, construct the Canonical Collection of LR(0) items.

3. Create the LR(0) parsing table and identify any conflicts. Comment whether the grammar is LR(0) or not.

4. Write the steps of the LR(0) parsing algorithm using a stack-based simulation for the string: `id + id * id`

5. What are the limitations of LR(0) parsing? Explain with an example grammar.

Programming Exercises (C++)

Problem 1: LR(0) Item Generation Write a C++ program to read a context-free grammar and generate LR(0) items.

Problem 2: LR(0) Canonical Collection Extend the program to generate the Canonical Collection of LR(0) items using closures and GOTO operations.

Explore the codes in GitHub
Click or Scan the QR

Chapter 9

Syntax trees and Directed Acyclic Graphs

In programming languages, expressions and statements follow a structured hierarchy that compilers analyze before generating executable code. A syntax tree represents this structure by breaking down code into operators and operands, organizing them in a tree-like format where each operation branches into its components. This helps compilers perform semantic checks and translate code efficiently.

However, real-world programs often contain repeated calculations, leading to redundancy. A Directed Acyclic Graph (DAG) optimizes this by merging identical subexpressions, ensuring each unique computation is stored only once. This optimization reduces memory usage and speeds up execution.

In this chapter, we'll explore how syntax trees capture program structure and how DAGs enhance efficiency by eliminating redundancy—key concepts in compiler design and code optimization.

9.1. Syntax trees

A syntax tree (or Abstract Syntax Tree) is a simplified version of a parse tree that shows only the important parts of code—like operators (e.g., +, *) and operands (e.g., variables, numbers)—while ignoring extra details like parentheses or grammar rules.

Parse Tree vs Syntax Tree

Aspect	Parse Tree	Syntax Tree
Purpose	Represents the grammatical structure of the input according to grammar rules	Represents the abstract syntactic structure, focusing on the meaning
Detail Level	Includes all grammar rules and non-terminals	Omits unnecessary grammar details (e.g., extra parentheses, some non-terminals)
Nodes	Contains both terminals and non-terminals	Contains only essential constructs (operators, operands)
Structure Size	Larger and more detailed	More compact
Use Case	Syntax analysis	Semantic analysis, optimization, code generation

Example:

Here is a grammar, generate parse tree and syntax tree

$$E \rightarrow E + T \mid T \quad T \rightarrow T * F \mid F \quad F \rightarrow (E) \mid id$$

Example input: `id + id * id`

Parse Tree
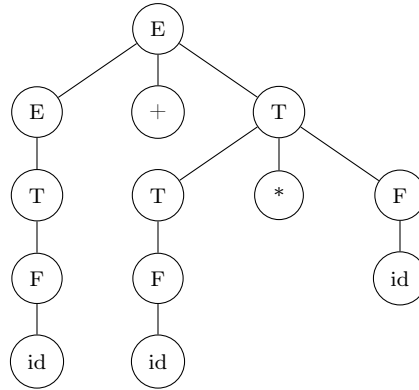
Syntax Tree
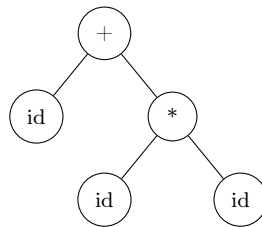
9.2. Directed Acyclic Graph

A **Directed Acyclic Graph (DAG)** is:

- An abstract syntax tree (AST) with a unique node for each value.

- A directed graph that contains no cycles.

Use of DAG for Optimizing Basic Blocks

- DAG is a useful data structure for implementing transformations on basic blocks.

- A basic block can be optimized by constructing a DAG.

- Transformations such as:

    - Common subexpression elimination

    - Dead code elimination

  can be applied using DAG.

- DAG is constructed from three-address statements.

Properties of DAG

- The reachability relation forms a partial order.

- Any finite partial order can be represented by a DAG.

- Transitive reduction and transitive closure are uniquely defined.

- Every DAG has a topological ordering.

Applications of DAG

DAG is used for:

1. Identifying common subexpressions.

2. Determining which names are used in the block and computed outside.

3. Identifying statements whose computed values are used outside the block.

4. Simplifying quadruples by eliminating redundant assignments.

Rules for DAG Construction

Rule 1: Leaf nodes represent identifiers or constants. Interior nodes represent operators.

Rule 2: Before creating a new node, check if a node with the same children already exists. If yes, reuse it to eliminate common subexpressions.

Rule 3: Assignments of the form $x_i = y$ should only be performed if necessary.

9.3. Examples of Directed Acyclic Graphs

Example 1: Construct DAG for $(a + b) \times (a + b + c)$

Three Address Code:

$$t_1 = a + b$$
$$t_2 = t_1 + c$$
$$t_3 = t_1 \times t_2$$

DAG for Expression: $(a + b) \times (a + b + c)$
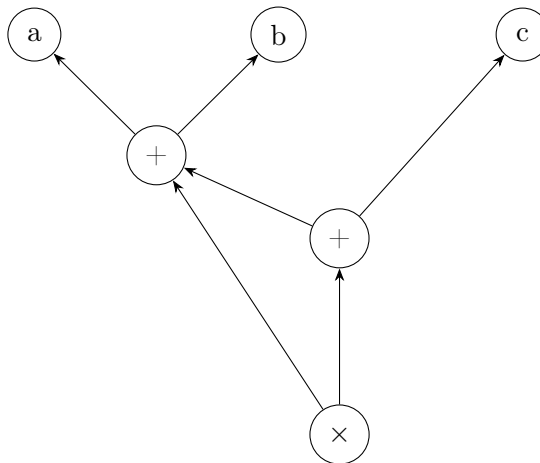
Example 2: Construct DAG for $\big(((a+a)+(a+a))+((a+a)+(a+a))\big)$

DAG for Expression:
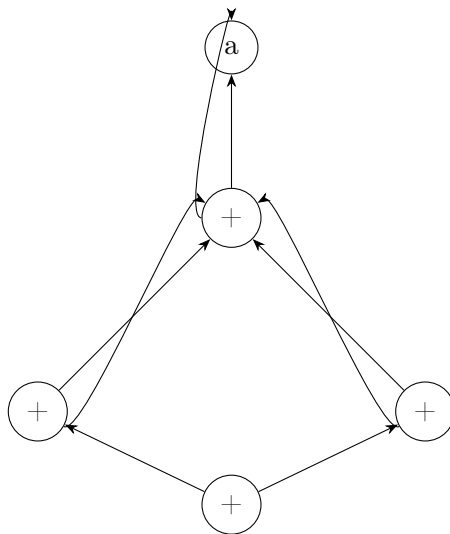
Example 3: Construct DAG for the Block

```
a = b * c
d = b
e = d * c
b = e
f = b + c
```

DAG for the Block:
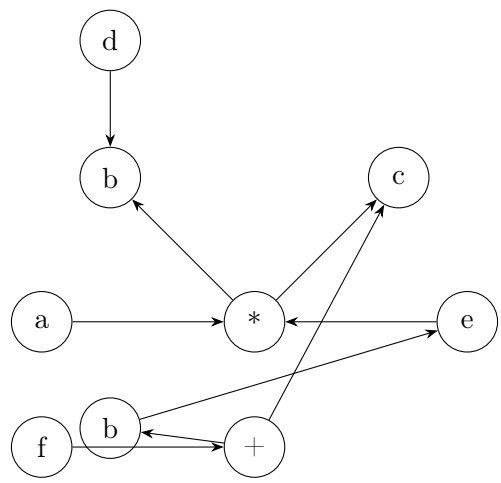
Problem 4: Optimize the Block from Problem 3

Optimized Block:

```
a = b * c
d = b
f = a + c
g = f + d
```

DAG for the optimized block:
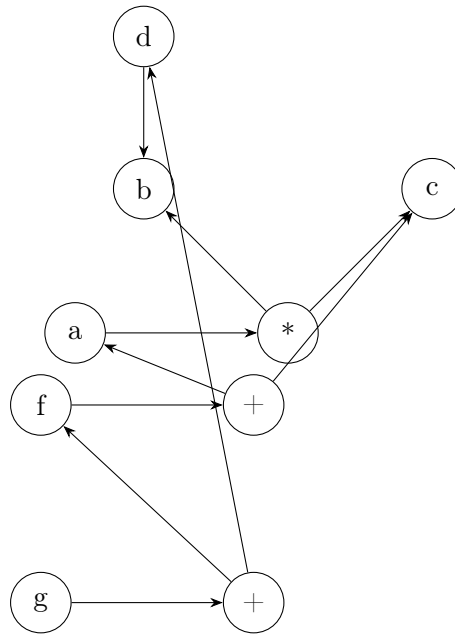
Problem 5: DAG Optimization of a Loop Block

```
L10: S1 = 4 * I
S2 = addy(A) - 4
S3 = S2[S]
S4 = 4 * I
S5 = addy(B) - 4
S6 = S5[S4]
S7 = S3 * S6
S8 = PROD + S7
PROD = S8
S9 = I + 1
I = S9
IF I <= 20 GOTO L10
```

DAG Optimization for Loop Block
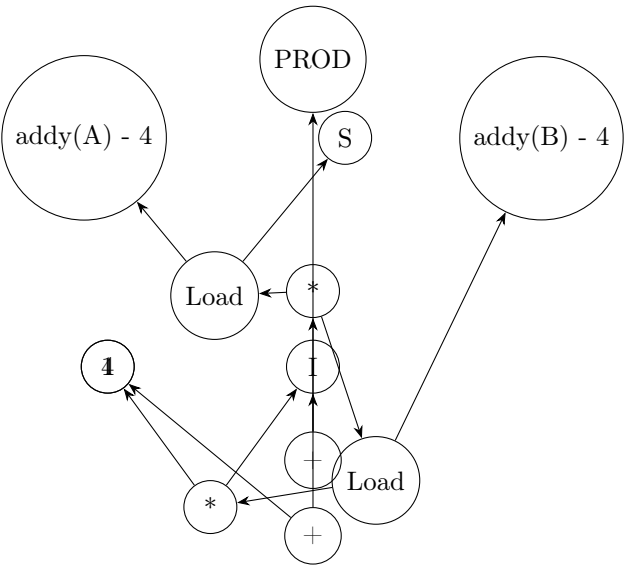
9.4. Problems and Exercises

9.4.1 Programming Problems

Problem 1: Write a program to Check if a Node is an Operator or Operand (e.g., $+ \rightarrow$ operator, $5 \rightarrow$ operand).

```
#include <stdio.h>
#include <ctype.h>
#include <string.h>

// Function to check if a string is operator
int isOperator(const char* str) {
    // For simplicity, assume operator is a single char from
        + - * /
    if (strlen(str) == 1) {
        char c = str[0];
        if (c == '+' || c == '-' || c == '*' || c == '/') {
            return 1;  // Operator
        }
    }
    return 0; // Operand
}

int main() {
    char node1[] = "+";
    char node2[] = "5";
    char node3[] = "x";
    char node4[] = "-";

    printf("%s is %s\n", node1, isOperator(node1) ? "Operator
        " : "Operand");
    printf("%s is %s\n", node2, isOperator(node2) ? "Operator
        " : "Operand");
    printf("%s is %s\n", node3, isOperator(node3) ? "Operator
        " : "Operand");
    printf("%s is %s\n", node4, isOperator(node4) ? "Operator
        " : "Operand");

    return 0;
}
```

Sample Input and Output

Input	Output
(No input)	+ is Operator
	5 is Operand
	x is Operand
	- is Operator

Problem 2: Write a program to Count Nodes in Syntax Tree: 2+3

```c
#include <stdio.h>
#include <stdlib.h>

typedef struct Node {
    char data;
    struct Node *left, *right;
} Node;

Node* createNode(char data) {
    Node* newNode = (Node*)malloc(sizeof(Node));
    newNode->data = data;
    newNode->left = newNode->right = NULL;
    return newNode;
}

int countNodes(Node* root) {
    if (root == NULL) return 0;
    return 1 + countNodes(root->left) + countNodes(root->
        right);
}

int main() {
    Node* root = createNode('+');
    root->left = createNode('2');
    root->right = createNode('3');

    printf("Total nodes: %d", countNodes(root)); // Output: 3
    return 0;
}
```

Sample Input and Output	
Input	**Output**
(No input needed since the tree is hardcoded)	Total nodes: 3

9.4.2   Exercises

1. Given the expression:

$$(x + y) * (x + y) + z$$

- Draw the syntax tree.

- Draw the DAG to demonstrate subexpression sharing.

2. For the arithmetic expression:

$$a * b + a * b + c$$

- Draw the syntax tree.

- Draw the DAG showing how redundant computations are merged.

3. Given this grammar:

$$\begin{cases} E \rightarrow E - T \mid T \\ T \rightarrow T/F \mid F \\ F \rightarrow (E) \mid id \end{cases}$$

- Draw the syntax tree for the expression: id - id / id

4. For the expression:

$$(a + b) + (a + b) * c$$

- Show both the syntax tree and the equivalent DAG.

5. Draw the **syntax tree** for:

$$(a + b) * (c + d)$$

- then convert it into the equivalent **DAG.**

Chapter 10

Three Address Code

10.1. Introduction to Intermediate Code

During the compilation of a high-level language, the source code is typically translated into an intermediate representation before generating the final machine code. One such representation is the **Three Address Code (TAC)**.

Role in Compilation

Intermediate code serves as a bridge between the source program and machine code. It abstracts away machine-specific details, making optimization and code generation simpler and more modular.

- Separates front-end (syntax analysis) and back-end (code generation).

- Provides a uniform format for optimization techniques.

- Facilitates portability and reusability of the compiler back-end.

Benefits of Intermediate Representation

The main advantages of using an intermediate representation like TAC include:

- Easier to analyze and transform than source code.

- Simpler structure than complex machine-level instructions.

- Ideal for performing compiler optimizations (e.g., constant folding, dead code elimination).

- Allows reuse of target code generators for multiple languages.

10.2. Definition of Three Address Code

Three Address Code (TAC) is a type of intermediate representation widely used in compilers. Each instruction in TAC contains at most three addresses or operands—typically two source operands and one destination. These operands can be variables, constants, or temporary variables generated during compilation.

Structure of TAC Instruction

A typical TAC instruction follows the format:

```
x = y op z
```

Where:

- `x` is the target or result variable.

- `y` and `z` are source operands.

- `op` is a binary operator such as `+`, `-`, `*`, or `/`.

In cases of unary operations or simple assignments, the format becomes simpler:

- `x = -y` (Unary operation)
- `x = y` (Assignment)

Why It's Called "Three Address"

The name "Three Address Code" comes from the fact that each instruction involves at most three addresses:

1. One address for the result.

2. Two addresses for operands.

This makes TAC more expressive than assembly language (which often supports only two-address formats), yet simpler and easier to analyze than full high-level source code.

Example

For the high-level expression:

$$a = (b + c) * d$$

The corresponding TAC would be:

```
t1 = b + c
t2 = t1 * d
a = t2
```

Here, `t1` and `t2` are temporary variables used to hold intermediate results.

10.3. Types of Three Address Code Instructions

Three Address Code (TAC) supports a variety of instruction types that are designed to represent common operations in a high-level language. These include assignment, arithmetic operations, control flow, procedure calls, and memory access.

Each TAC instruction is simple, typically involving no more than one operator and at most three operands.

10.3.1 Assignment Statements

Assignment in TAC involves copying the value of one variable to another.

```
x = y
```

This is a direct assignment where the value of `y` is stored in `x`.

Example:

```
a = b
```

10.3.2 Arithmetic and Logical Operations

Binary operations such as addition, subtraction, multiplication, division, and logical operators are commonly used in TAC.

```
x = y + z
x = y - z
x = y * z
x = y / z
x = y && z
x = y || z
```

Example:

For the high-level code: `a = b * c + d`

TAC will be:

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

10.3.3 Unary Operations

TAC also supports unary operations such as negation, logical not, and address referencing.

```
x = -y
x = !y
x = &y   ; address of y
```

Example:

```
t1 = -b
a = t1
```

10.3.4   Conditional and Unconditional Jumps

TAC supports control flow through conditional and unconditional jump instructions.

- **Unconditional jump:**

    ```
    goto L1
    ```

- **Conditional jump:**

    ```
    if x < y goto L2
    if x == 0 goto L3
    ```

- **Label definition:**

    ```
    L1:
    ```

Example:

For the code:

```
if (a > b)
    x = 1;
else
    x = 0;
```

TAC:

```
if a <= b goto L1
x = 1
goto L2
L1: x = 0
L2:
```

10.3.5   Procedure Calls and Parameters

TAC supports function and procedure calls using the following instructions:

```
param x       ; push parameter x
call p, n    ; call procedure p with n parameters
x = call p, n ; call and store return value in x
```

Example:

For the code:

```
z = max(a, b)
```

TAC:

```
param a
param b
z = call max, 2
```

10.3.6   Address and Pointer Instructions

TAC allows indirect memory access similar to pointer dereferencing and value assignment.

```
x = *y       ; x gets value at address y
*y = x       ; value x is stored at address y
```

Example:

If `ptr` holds the address of variable `a`, then:

*ptr = 10 ; assign 10 to the memory location pointed by ptr

TAC:

```
t1 = 10
*ptr = t1
```

10.4. Use of Temporary Variables

In Three Address Code (TAC), **temporary variables** are used to store intermediate results during the translation of complex expressions. These variables are not part of the original source code but are introduced by the compiler to simplify computation and preserve intermediate values.

10.4.1   Purpose of Temporaries

Temporary variables help in:

- Breaking down complex expressions into simpler components.

- Avoiding repeated computation of subexpressions.

- Managing the order of evaluation in expressions.

- Facilitating code optimization.

Why Are They Needed?

High-level expressions may involve nested operations that cannot be directly represented in a single TAC instruction. Temporaries are introduced to sequentially compute each subexpression.

10.4.2   Naming Convention

Temporary variables are typically named using a prefix like `t` followed by a unique number (e.g., `t1`, `t2`, `t3`, etc.). This makes them easy to generate and track during code generation.

Example 1: Arithmetic Expression

Given:

```
a = (b + c) * (d - e)
```

TAC using temporaries:

```
t1 = b + c
t2 = d - e
t3 = t1 * t2
a = t3
```

Example 2: Logical and Comparison Expression

Given:

```
x = (a < b) || (c > d)
```

TAC:

```
t1 = a < b
t2 = c > d
x = t1 || t2
```

Example 3: Function Call with Expression

Given:

```
z = f(a + b, c * d)
```

TAC:

```
t1 = a + b
t2 = c * d
param t1
param t2
z = call f, 2
```

10.5. Generating TAC for Expressions

Converting high-level expressions into Three Address Code (TAC) is a crucial step in compiler design. The goal is to transform complex expressions into a sequence of simple, low-level TAC instructions using temporary variables.

10.5.1   Arithmetic Expressions

Arithmetic expressions often require evaluation in a specific order due to operator precedence and associativity. TAC uses temporary variables to maintain this order.

Example 1: Simple Arithmetic

Given:

```
a = b + c * d
```

TAC:

```
t1 = c * d
t2 = b + t1
a = t2
```

Example 2: Nested Arithmetic

Given:

```
a = (b + c) * (d - e)
```

TAC:

```
t1 = b + c
t2 = d - e
t3 = t1 * t2
a = t3
```

10.5.2   Boolean Expressions

Boolean expressions use logical operators (, ||, !) and comparison operators (==, !=, <, >, <=, >=). These can also be translated into TAC.

Example 3: Boolean Logic

Given:

```
x = (a < b) && (c > d)
```

TAC:

```
t1 = a < b
t2 = c > d
x = t1 && t2
```

10.5.3   Array and Pointer Expressions

Accessing arrays or using pointers involves address computations, which can be expressed in TAC using index and dereference notation.

Example 4: Array Access

Given:

```
a = arr[i] + 5
```

TAC:

```
t1 = arr[i]
t2 = t1 + 5
```

```
a = t2
```

Example 5: Pointer Dereferencing

Given:

```
*x = y + z
```

TAC:

```
t1 = y + z
*x = t1
```

10.5.4   Operator Precedence and Parentheses

The compiler must honor the operator precedence and parentheses while generating TAC.

Example 6: With Parentheses

Given:

```
x = (a + b) / (c - d)
```

TAC:

```
t1 = a + b
t2 = c - d
x = t1 / t2
```

10.6. TAC for Control Flow Statements

Control flow statements like `if`, `while`, and `for` are essential in programming languages. These constructs are translated into Three Address Code (TAC) using conditional jumps, labels, and gotos to simulate flow of control.

If-Else Statement

An `if-else` statement checks a condition and executes one of two branches. The condition is translated using a conditional jump, and labels mark the true, false, and exit paths.

High-Level Code:

```
if (a < b)
    x = 1;
else
    x = 0;
```

TAC:

```
if a >= b goto L1
x = 1
goto L2
L1: x = 0
L2:
```

While Loop

A `while` loop executes a block repeatedly as long as a condition holds true. The condition is placed before the loop body.

High-Level Code:

```
while (i < 10) {
    i = i + 1;
}
```

TAC:

```
L1: if i >= 10 goto L2
t1 = i + 1
i = t1
```

```
goto L1
L2:
```

For Loop

A `for` loop can be broken into initialization, condition, increment, and body. TAC represents this by using labels and jumps.

High-Level Code:

```
for (i = 0; i < 5; i = i + 1) {
    sum = sum + i;
}
```

TAC:

```
i = 0
L1: if i >= 5 goto L2
t1 = sum + i
sum = t1
t2 = i + 1
i = t2
goto L1
L2:
```

Switch Statement (Optional)

A `switch` can be implemented as a sequence of if-goto statements or using a jump table.

High-Level Code:

```
switch (x) {
    case 1: y = 100; break;
    case 2: y = 200; break;
    default: y = 0;
```

```
}
```

TAC:

```
if x != 1 goto L1
y = 100
goto L4
L1: if x != 2 goto L2
y = 200
goto L4
L2: y = 0
L4:
```

10.7. Backpatching and Flow Control

In some cases during the translation of control flow statements (like `if`, `while`, or boolean expressions), the exact target of a jump is not known immediately. **Backpatching** is a technique that allows us to generate jumps with unknown labels and later fill in those labels when their addresses become known.

10.7.1 Need for Backpatching

Backpatching is necessary when:

- Control flow has forward jumps, such as `goto L1`, but the label `L1` is defined later.

- Boolean expressions with short-circuit evaluation (e.g., `A || B`, `A && B`) need to jump based on runtime conditions.

Rather than tracking jump targets manually, backpatching maintains lists of incomplete jumps which are updated once the target is defined.

10.7.2 Implementing Backpatching

We use three basic procedures in backpatching:

- `makelist(i)`: creates a new list containing index `i`.

- `merge(p1, p2)`: combines two lists into one.

- `backpatch(p, L)`: inserts label `L` into all instructions on list `p`.

Example (Boolean Expression):

Given:

```
if (a < b || c > d) x = 1;
```

We translate it as:

```
if a < b goto L1
if c <= d goto L2
L1: x = 1
L2:
```

But while translating, we don't know where `L1` and `L2` are. So we:

- Emit conditional jumps with unknown targets.

- Save their instruction indices in lists.

- After we reach the target label, use `backpatch()` to fill in those targets.

10.7.3   Backpatching in While Loop

While loop:

```
while (a < b) {
    S;
}
```

We generate:

```
L1: if a >= b goto L2   ; condition check
    [code for S]        ; body
    goto L1             ; repeat
```

```
L2:                         ; exit point
```

Here,

- `L1` is the beginning of the loop.

- `L2` is the backpatched exit point.

10.8. Three Address Code Representations

Three address code can be implemented as a record with address fields. There are three common representations:

- Quadruples

- Triples

- Indirect Triples

1. Quadruples

Each instruction is divided into four fields:

- **op**: Operator code

- **arg1**, **arg2**: Operands

- **result**: Result of the expression

Exceptions:

- Unary operation: $x = op\ y$ is represented with `op` in operator, `y` in arg1, and `x` in result. `arg2` is unused.

- Parameter passing: `param t1` uses `param` in operator and `t1` in arg1. `arg2` and result are unused.

- Jump statements: Target label is placed in the result field.

2. Triples

Triples avoid temporary variables by referencing instruction positions directly.

3. Indirect Triples

Indirect triples use an instruction array to point to triples in desired order, allowing optimizers to rearrange sub-expressions freely.

Illustration 1: Expression Translation

Expression: $a + b \times c/c + b \times a$

Three Address Code:

$$T_1 = b \times c$$
$$T_2 = T_1/c$$
$$T_3 = b \times a$$
$$T_4 = a + T_2$$
$$T_5 = T_4 + T_3$$

Quadruple Representation

Location	op	arg1	arg2	result
0	*	b	c	T1
1	/	T1	c	T2
2	*	b	a	T3
3	+	a	T2	T4
4	+	T4	T3	T5

Triple Representation

Location	op	arg1	arg2
0	*	b	c
1	/	(0)	c
2	*	b	a
3	+	a	(1)
4	+	(3)	(2)

Indirect Triple Representation

Statement	Location	op	arg1	arg2
35	0	*	b	c
36	1	/	(0)	c
37	2	*	b	a
38	3	+	a	(1)
39	4	+	(3)	(2)

Illustration 2: Expression Translation

Expression: $a = b \times (-c) + b \times (-c)$

Three Address Code:

$$T_1 = -c$$
$$T_2 = b \times T_1$$
$$T_3 = -c$$
$$T_4 = b \times T_3$$
$$T_5 = T_2 + T_4$$
$$a = T_5$$

Quadruple Representation

Location	op	arg1	arg2	result
0	uminus	c	-	T1
1	*	b	T1	T2
2	uminus	c	-	T3
3	*	b	T3	T4
4	+	T2	T4	T5
5	=	T5	-	a

Triple Representation

Location	op	arg1	arg2
0	uminus	c	-
1	*	b	(0)
2	uminus	c	-
3	*	b	(2)
4	+	(1)	(3)
5	=	(4)	a

Indirect Triple Representation

Statement	Location	op	arg1	arg2
36	0	uminus	c	-
37	1	*	b	(0)
38	2	uminus	c	-
39	3	*	b	(2)
40	4	+	(1)	(3)
41	5	=	(4)	a

10.9. Problems and Exercises

10.9.1   Programming Problems

Problem 1: Generate the three address code for the arithmetic expression:
`a = (b + c) * (d - e)`.

```
// Sample C++-like Pseudocode
t1 = b + c
t2 = d - e
t3 = t1 * t2
a = t3
```

Sample Input and Output

Input	Output
a = (b + c) * (d - e)	t1 = b + c
	t2 = d - e
	t3 = t1 * t2
	a = t3

Problem 2: Generate TAC for the following conditional statement: `if (a > b) x = y + z;`

```
// Sample Pseudocode
if a <= b goto L1
t1 = y + z
x = t1
L1:
```

Sample Input and Output

Input	Output
if (a > b) x = y + z;	if a <= b goto L1
	t1 = y + z
	x = t1
	L1:

Problem 3: Generate Three Address Code for the following while loop:
`while (i < 10) { i = i + 1; }`

```
// Sample TAC
L1:
if i >= 10 goto L2
t1 = i + 1
i = t1
goto L1
L2:
```

Sample Input and Output

Input	Output
while (i < 10) { i = i + 1; }	L1:
	if i >= 10 goto L2
	t1 = i + 1
	i = t1
	goto L1
	L2:

10.9.2 Exercises

Academic Exercises: Three Address Code

1. Convert the expression `a + b * (c - d)` to three address code.

2. Generate TAC for nested if-else conditions.

3. Generate TAC for:

```
if (x < y)
    if (y < z)
        max = z;
    else
        max = y;
```

4. Generate TAC for switch-case statements.

5. Write TAC for a function call with multiple parameters.

Explore the codes in GitHub
Click or Scan the QR

Chapter 11

Basic Blocks and Flow Graph

A basic block is a set of code lines where the flow of execution goes straight from the start to the end, without any jumps or breaks in between. It has only one way in and one way out. Breaking a program into basic blocks makes it easier for the compiler to analyze and improve the code. To understand how these blocks connect, we use a flow graph — a diagram that shows the possible paths the program can take during execution.

In this chapter, we'll learn what basic blocks are, how to identify them using leader rules, and how to draw a flow graph by following some simple steps.

11.1. Basic Blocks

A basic block is a group of instructions that run one after another, without any jumps or stops in between. Once the first instruction starts, all the others follow in order. This makes it easier for the compiler to understand and improve the code during analysis.

Breaking Intermediate Code into Basic Blocks

To break a program into basic blocks, we first find the special starting points called leaders and then group the instructions based on them. Here's how we do it:

How to Find Leaders:

- Rule 1: The very first 3-address code instruction is always a leader.

- Rule 2: Any instruction that a jump goes to (either conditional or unconditional) is also a leader.

- Rule 3: The instruction that comes right after a jump is also marked as a leader.

How to Form Basic Blocks:

- A basic block starts from a leader and includes all the following instructions until the next leader is found.

- The block that starts with the first leader is called the Initial Block.

11.2. Understanding the Flow Graph

After breaking the program into basic blocks, we use a flow graph to show how the control moves from one block to another during execution. In this graph, each node represents a basic block, and arrows (edges) show the direction of flow between them.

Flow Graph Rules:

1. Arrows (edges) show how control passes from one block to another.

2. If a block doesn't end with a jump, it automatically connects to the next block.

3. If there's a `goto` or branch, an arrow is drawn from that block to the target block, showing the jump in control.

Example of Basic Blocks and Flow Graph:

Consider the following intermediate code:

```
1)   i = m - 1
2)   j = n
3)   t1 = 4 * n
4)   v = a[t1]
5)   i = i + 1
6)   t2 = 4 * i
7)   t3 = a[t2]
8)   if t3 < v (goto 5)
9)   j = j - 1
10)  t4 = 4 * j
11)  t5 = a[t4]
12)  if t5 > v (goto 9)
13)  if i >= j (goto 15)
14)  t6 = 4 * i
```

Selecting the Leaders from 3-Address Code

```
1)    i = m - 1                          {L1}
2)    j = n
3)    t1 = 4 * n
4)    v = a[t1]
5)    i = i + 1                          {L2}
6)    t2 = 4 * i
7)    t3 = a[t2]
8)    if t3 < v (goto 5)
9)    j = j - 1                          {L2, L3}
10)   t4 = 4 * j
11)   t5 = a[t4]
12)   if t5 > v (goto 9)
13)   if i >= j (goto 15)                {L3}
14)   t6 = 4 * i                         {L3}
```

Here L1, L2, and L3 denote leaders, and the number beside them represents

the rule number used to identify them. A single instruction can be a leader by multiple rules.

Now Constructing the Basic Block and Flow Graph

BASIC BLOCK:

B1
1)	$i = m - 1$
2)	$j = n$
3)	$t1 = 4 * n$
4)	$v = a[t1]$

B2
5)	$i = i + 1$
6)	$t2 = 4 * i$
7)	$t3 = a[t2]$
8)	if $t3 \leq v$ (goto 5)

B3
9)	$j = j - 1$
10)	$t4 = 4 * j$
11)	$t5 = a[t4]$
12)	if $t5 \geq v$ (goto 9)

B4
13)	if $i \geq j$ (goto 15)

B5
14)	$t6 = 4 * i$

Figure 10.1: Basic Block

FLOW GRAPH:
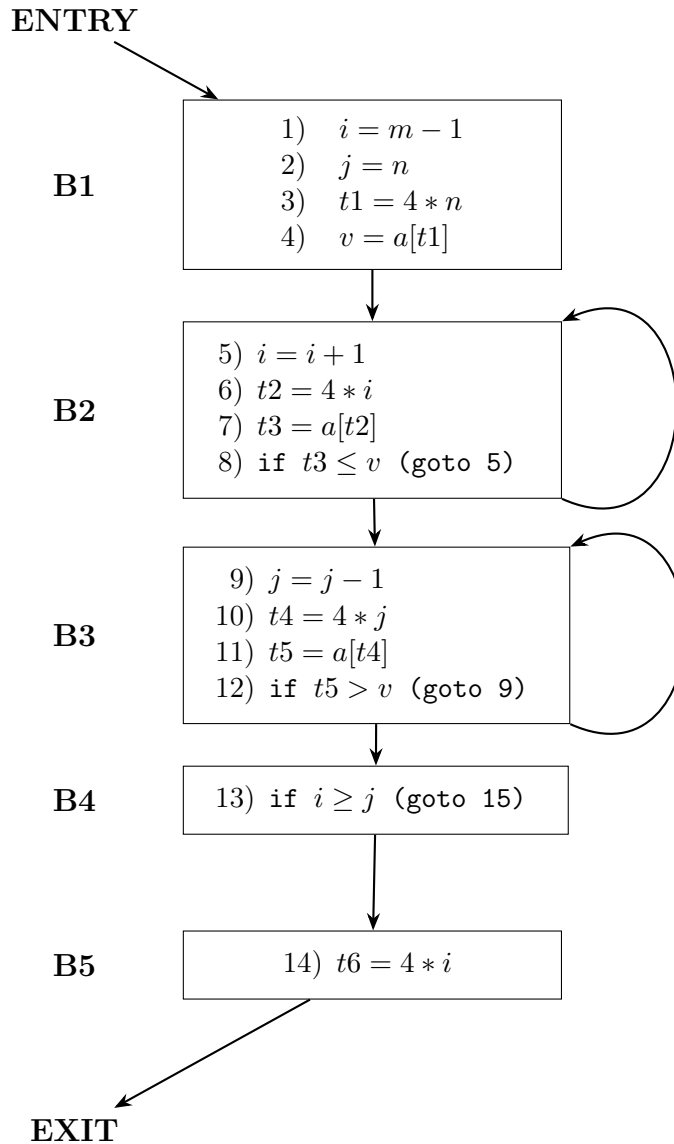
Figure 10.2: Flow Graph

11.3. Problems and Exercises

11.3.1 Programming Problems

Problem 1: Given a C++ code snippet, identify the basic blocks and draw the flow graph.

```
#include<iostream>
using namespace std;
int main() {
    int a = 5, b = 10;
    if (a < b) {
        a = a + b;
    } else {
        b = b - a;
    }
    cout << a << " " << b;
    return 0;
}
```

Sample Input and Output	
Input	**Output**
N/A	Basic Blocks:
	B1: int a = 5, b = 10;
	B2: if (a < b)
	B3: a = a + b;
	B4: b = b - a;
	B5: cout « a « " " « b; return 0;

Problem 2: Construct the control flow graph (CFG) of the following code.

```
int x = 0;
for (int i = 0; i < 5; i++) {
    x = x + i;
    if (x > 10) {
        break;
    }
}
```

Sample Input and Output

Input	Output
N/A	Basic Blocks:
	B1: int x = 0; int i = 0;
	B2: i < 5
	B3: x = x + i
	B4: x > 10
	B5: break
	B6: i++
	Flow Graph: Edges: B1 → B2 → B3 → B4 → (B5 or B6) → B2

Problem 3: Break the following code into basic blocks and draw the flow graph.

```
int a = 1, b = 2;
if (a == b) {
    a += b;
} else if (a > b) {
    a -= b;
} else {
    b -= a;
}
return a + b;
```

Sample Input and Output

Input	Output
N/A	Basic Blocks:
	B1: int a = 1, b = 2;
	B2: if (a == b)
	B3: a += b;
	B4: else if (a > b)
	B5: a -= b;
	B6: else
	B7: b -= a;
	B8: return a + b;
	Flow Graph: B1 → B2 → B3/B4 → B5/B6 → B7 → B8

11.3.2   Exercises

Academic Exercises

1. Identify leaders from a sequence of TAC instructions.

2. Construct a flow graph from given basic blocks.

3. Analyze loops and dominators in a control flow graph.

4. Explain the role of basic blocks in optimization.

Programming Exercise 1: Leader Identification

Input:

```
a = b + c
if a < d goto L1
e = a - d
goto L2
L1: e = a + d
L2: print e
```

Output:

```
Leaders:
1: a = b + c
2: L1
3: L2

Basic Blocks:
Block 1:
a = b + c
if a < d goto L1

Block 2:
e = a - d
goto L2

Block 3:
L1: e = a + d
```

```
Block 4:
L2: print e
```

Programming Exercise 2: Control Flow Graph

Flow Graph (Adjacency List):

```
Block 1 -> Block 2, Block 3
Block 2 -> Block 4
Block 3 -> Block 4
Block 4 -> (end)
```

Explore the codes in GitHub
Click or Scan the QR

Chapter 12

Code Optimization

Code optimization is a fundamental phase in the backend of a compiler where the intermediate code is transformed to improve performance and efficiency. The goal is to generate a semantically equivalent program that consumes fewer resources, such as CPU time, memory, or power, without changing the output of the program.

Why is Optimization Important?

Optimized code is crucial in real-world applications where performance, energy consumption, and storage space are significant concerns. For example:

- In embedded systems, optimization reduces power usage and increases speed.

- In mobile apps, it ensures faster response and battery efficiency.

- In scientific applications, it helps reduce runtime for large-scale computations.

Objectives of Code Optimization

The main objectives are:

1. **Minimize Execution Time:** Make the program run faster by reducing instruction count or complexity.

2. **Reduce Memory Usage:** Save RAM or storage by optimizing data and instructions.

3. **Decrease Power Consumption:** Important for mobile and embedded devices.

4. **Maintain Program Semantics:** The output must remain unchanged.

5. **Enhance Code Maintainability and Readability:** Some optimizations may simplify code.

When Does Optimization Occur?

Optimization can be performed at various stages of compilation:

- **Early Optimization:** During intermediate code generation.

- **Late Optimization:** Just before code generation (machine-specific).

- **Link-time Optimization (LTO):** During linking phase.

- **Run-time Optimization:** In Just-In-Time (JIT) compilation.

Static vs Dynamic Optimization

Static Optimization: Performed during compile-time. Examples include dead code elimination, constant folding, etc.

Dynamic Optimization: Done at runtime using profiling information. Used in Java JIT compilers or modern virtual machines.

Safe vs Speculative Optimization

- **Safe Optimization:** Guaranteed not to change the behavior of the program (e.g., removing dead code).

- **Speculative Optimization:** Based on assumptions, may roll back if those assumptions are violated (e.g., branch prediction).

12.1. Types of Optimizations

Compiler optimizations are generally classified into two categories:

12.1.1 Machine-Independent Optimization

These optimizations are applied to the intermediate representation (IR) of the code and are not dependent on the target machine. They focus on improving the logical structure and efficiency of the program regardless of hardware architecture.

Examples:

- **Constant Folding:** Evaluate constant expressions at compile time.

- **Constant Propagation:** Replace variables with known constant values.

- **Common Subexpression Elimination (CSE):** Remove redundant calculations.

- **Dead Code Elimination:** Remove code that does not affect output.

- **Loop Optimization:** Improve performance by unrolling, hoisting invariants, or reducing induction variables.

Advantages:

- Portable across platforms.

- Generally safe and predictable.

- Simplifies code before hardware-specific generation.

12.1.2 Machine-Dependent Optimization

These optimizations depend on the architecture of the target machine (e.g., number of registers, instruction set, CPU pipeline behavior). They are ap-

plied after or during code generation and directly impact how instructions are selected and scheduled.

Examples:

- **Register Allocation:** Assign variables to machine registers efficiently.

- **Instruction Scheduling:** Reorder instructions to reduce pipeline stalls.

- **Peephole Optimization:** Replace small sequences of machine instructions with more efficient equivalents.

- **Instruction Selection:** Choose the most optimal hardware instructions.

Advantages:

- Direct performance improvement based on hardware.

- Makes use of CPU features like SIMD, parallelism, and instruction pipelining.

Comparison Table

Machine-Independent Optimization	Machine-Dependent Optimization
Works on Intermediate Representation (IR)	Works on Target Machine Code
Portable across different architectures	Specific to a particular architecture
Focuses on logic-level improvements	Focuses on hardware-level improvements
Examples: Loop optimization, CSE, Dead code removal	Examples: Register allocation, Instruction scheduling

12.2. Basic Blocks and Flow Graphs

A **basic block** is a sequence of consecutive statements in a program with the following properties:

- The flow of control enters at the beginning of the block and exits at the end.

- There are no jump or halt statements in the middle of the block.

- The code inside a basic block always executes sequentially.

Example:

Consider the following code fragment:

```
1. a = b + c
2. d = a - e
3. if d > 0 goto L1
4. f = d * 2
5. e = f - 1
```

From this code, we can extract the basic blocks as:

- **B1:** Line 1, 2

- **B2:** Line 3

- **B3:** Line 4, 5

Leaders and Construction of Basic Blocks

To identify basic blocks in a program, we first identify the **leaders**. A leader is the first instruction of a basic block. The following rules define leaders:

1. The first statement is always a leader.

2. Any statement that is the target of a conditional or unconditional jump is a leader.

3. Any statement that immediately follows a jump or branch statement is a leader.

Once the leaders are identified, each leader begins a basic block and includes all statements up to (but not including) the next leader.

Control Flow Graph (CFG)

A **Control Flow Graph (CFG)** is a directed graph where:

- Each node represents a basic block.

- There is an edge from block B_i to block B_j if the control can flow from B_i to B_j during program execution.

Example CFG for previous code:

- Block B1 leads to B2 (normal flow)

- Block B2 can go to B3 (if condition false) or jump to label L1

- Block B3 is the continuation of flow

You can visualize this as:
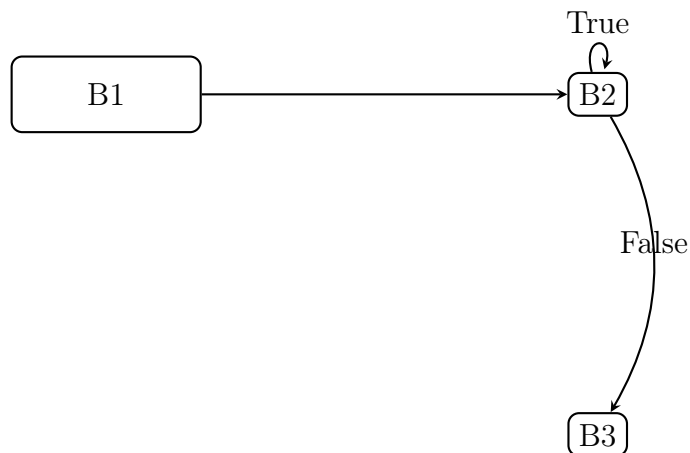
Note: Label L1 and additional blocks may be shown for larger CFGs.

Predecessors and Successors

- A block B_j is a **successor** of B_i if control can flow from B_i to B_j.

- Similarly, B_i is a **predecessor** of B_j.

These relationships are essential for performing data-flow analysis and optimization, as they help determine the flow of variables and possible execution paths.

12.3. Optimization Techniques

This section explores various optimization techniques used to improve program performance, reduce code size, and eliminate redundancy during code generation.

12.3.1 Constant Folding and Propagation

Constant folding is a technique where constant expressions are evaluated at compile time. **Constant propagation** replaces variables that have constant values with those constants.

- Example:

```
int x = 10;
int y = 20;
int z = x + y;  // Replaced with z = 30
```

12.3.2   Dead Code Elimination

Dead code refers to code that is never executed or whose result is never used. Removing such code reduces program size and improves efficiency.

- Example:

```
int a = 5;
a = 10;   // The first assignment is dead
```

12.3.3   Strength Reduction

This replaces expensive operations with cheaper ones.

- Example:

```
x = y * 2;   // Replace with: x = y + y;
```

12.3.4   Common Subexpression Elimination

If an expression is computed more than once and the variables involved have not changed, it can be computed once and reused.

- Example:

```
a = b + c;
d = b + c;   // Reuse the value of a instead
```

12.3.5   Copy Propagation

This optimization replaces variables that are assigned the value of another variable with the original variable.

- Example:

```
a = b;
c = a + 1;   // Replace with: c = b + 1;
```

12.3.6   Loop Invariant Code Motion

Moves computations that yield the same result on each loop iteration outside the loop.

- Example:

```
for (int i = 0; i < n; i++) {
    x = y + z;   // Move outside if y and z do not change
}
```

12.3.7   Induction Variable Elimination

Simplifies or removes induction variables (variables that change systematically in loops) to improve efficiency.

- Example:

```
i = 0;
t = 4 * i;
while (i < n) {
    ...
    i++;
    t = 4 * i;
}
```

Can be optimized using a separate counter.

12.3.8   Peephole Optimization

Performs small localized optimizations on a short sequence of target code instructions (the "peephole").

- Example:

```
MOV R1, #0
ADD R1, R1, R2
```

Can be optimized to:

```
MOV R1, R2
```

12.3.9   Code Motion

Code motion is a technique that moves computations out of loops or from frequently executed paths to less frequently executed ones, provided that doing so does not affect program semantics.

- Example:

```
for (int i = 0; i < n; i++) {
    t = a + b;
    arr[i] = t * i;
}
```

The expression `t = a + b` can be moved outside the loop:

```
t = a + b;
for (int i = 0; i < n; i++) {
    arr[i] = t * i;
}
```

12.3.10   Algebraic Simplification

Algebraic simplification replaces complex or redundant arithmetic expressions with simpler and more efficient equivalents, based on mathematical identities.

- Example:

```
x = x * 1;   // Simplified to: x = x;
y = y + 0;   // Simplified to: y = y;
z = z * 0;   // Simplified to: z = 0;
```

12.3.11   Unreachable Code Elimination

This technique removes code statements that will never be executed, often due to control flow statements like `return`, `break`, or `goto`.

- Example:

```
int func() {
    return 5;
    int x = 10;  // Unreachable
}
```

The line `int x = 10;` is unreachable and should be removed.

12.4. Data Flow Analysis

Data Flow Analysis (DFA) is a technique used by compilers to gather information about the possible set of values calculated at various points in a program. It is essential for many optimization techniques such as constant propagation, dead code elimination, and register allocation.

Data flow analysis is generally performed on the Control Flow Graph (CFG)

of a program by associating data flow information with each basic block.

Reaching Definitions

A **definition** of a variable is a statement that assigns a value to it. A definition *reaches* a point in the program if there exists a path from the definition to that point without any intervening redefinition of the variable.

- **Purpose:** To determine which definitions may reach a given program point.

- **Application:** Useful for optimizations like constant propagation and dead code elimination.

Live Variable Analysis

A variable is **live** at a point if its value will be used in the future before being redefined.

- **Purpose:** To find variables whose values are needed and those that can be safely discarded.

- **Application:** Helps in register allocation and dead code elimination.

Available Expression

An **available expression** at a program point is an expression that has already been computed and whose operands have not been modified since.

- **Purpose:** To avoid recomputation of expressions.

- **Application:** Used in common subexpression elimination.

Register Allocation and Assignment

- Need for Register Allocation
- Graph Coloring Technique

- Spilling

12.5. Loop Optimization Techniques

Loops are often the most time-consuming parts of a program. Optimizing loops can significantly improve performance.

12.5.1   Loop Unrolling

Loop unrolling reduces the overhead of loop control by replicating the loop body multiple times.

- Example:

```
for (int i = 0; i < 4; i++) {
    a[i] = a[i] * 2;
}
```

  Unrolled version:

```
a[0] = a[0] * 2;
a[1] = a[1] * 2;
a[2] = a[2] * 2;
a[3] = a[3] * 2;
```

12.5.2   Loop Fission and Fusion

Loop Fusion merges two adjacent loops with the same iteration space into one to reduce loop overhead.

Loop Fission splits a large loop into smaller loops, which may improve cache performance or enable parallelism.

- Example (Fusion):

```
for (int i = 0; i < n; i++) {
    a[i] = b[i] + 1;
}
for (int i = 0; i < n; i++) {
    c[i] = a[i] * 2;
}
```

Fused:

```
for (int i = 0; i < n; i++) {
    a[i] = b[i] + 1;
    c[i] = a[i] * 2;
}
```

12.5.3   Loop Interchange

Loop interchange swaps the order of nested loops to improve memory access patterns and cache performance.

- Example:

```
for (int i = 0; i < n; i++) {
    for (int j = 0; j < m; j++) {
        A[j][i] = ...;
    }
}
```

Interchanged loops:

```
for (int j = 0; j < m; j++) {
    for (int i = 0; i < n; i++) {
        A[j][i] = ...;
    }
}
```

12.5.4   Loop Invariant Code Motion (Revisited)

Loop invariant code motion moves computations that do not change inside the loop to outside to avoid redundant execution.

- Example:

```
for (int i = 0; i < n; i++) {
    int t = a + b;  // invariant, move outside loop
    arr[i] = t * i;
}
```

12.6. Example Programs and Step-by-Step Optimizations

This section demonstrates the application of various optimization techniques using Three Address Code (TAC). We will compare the code before and after optimization to understand the improvements clearly.

Using Three Address Code (TAC)

Consider the following TAC fragment:

```
1: t1 = a + b
2: t2 = t1 * c
3: t3 = a + b
4: t4 = t3 * d
5: t5 = t2 + t4
```

Step-by-Step Optimizations

- **Common Subexpression Elimination:** The expression `a + b` is computed twice (lines 1 and 3). We can eliminate the redundant com-

putation.

- **Copy Propagation:** Replace uses of variables that are copies of others.

- **Constant Folding:** If any constants are involved, evaluate at compile-time.

Optimized TAC Code

After applying common subexpression elimination and copy propagation, the code becomes:

```
1: t1 = a + b
2: t2 = t1 * c
3: t4 = t1 * d
4: t5 = t2 + t4
```

Comparison:

Before Optimization	After Optimization
1: t1 = a + b 2: t2 = t1 * c 3: t3 = a + b 4: t4 = t3 * d 5: t5 = t2 + t4	1: t1 = a + b 2: t2 = t1 * c 3: t4 = t1 * d 4: t5 = t2 + t4

This optimization reduces redundant computations and improves code efficiency.

Dead Code Elimination and Strength Reduction

Consider the following TAC code fragment:

```
1: a = b + c
2: d = a * 2
3: e = d + 0
```

```
4: f = e * 1
5: g = 10
6: h = g + 5
7: return h
8: i = 100    // Dead code: never used
```

Step 1: Dead Code Elimination

The instruction on line 8 assigns a value to `i` which is never used later, so it can be removed.

Step 2: Strength Reduction and Algebraic Simplification

- Multiplication by 2 can be replaced with addition: `d = a * 2` becomes `d = a + a`.

- Multiplication by 1 and addition of 0 are redundant and can be removed.

Optimized TAC code:

```
1: a = b + c
2: d = a + a
6: h = 10 + 5
7: return h
```

Explanation:

- Line 3 and 4 removed because they do not change the value.

- Line 5 combined into line 6 as constant addition.

- Line 8 removed as dead code.

Comparison:

Before Optimization	After Optimization
1: a = b + c	
2: d = a * 2	
3: e = d + 0	1: a = b + c
4: f = e * 1	2: d = a + a
5: g = 10	6: h = 10 + 5
6: h = g + 5	7: return h
7: return h	
8: i = 100	

12.7. Problems and Exercises

12.7.1 Programming Problems

Problem 1: Common Subexpression Elimination

```
int a = x * y;
int b = x * y + z;
```

Optimized Code:

```
int t = x * y;
int a = t;
int b = t + z;
```

Sample Input and Output	
Input	**Output**
x = 2, y = 3, z = 4	a = 6, b = 10

Problem 2: Loop Invariant Code Motion

```
for (int i = 0; i < n; i++) {
    int k = a * b;
    arr[i] = k + i;
}
```

Optimized Code:

```
int k = a * b;
for (int i = 0; i < n; i++) {
    arr[i] = k + i;
}
```

Input	Output
a = 2, b = 3, n = 5	arr = [6,7,8,9,10]

Problem 3: Dead Code Elimination

```
int a = 5;
int b = 10;
int c = a + b;
a = 20;
cout << c << endl;
```

Optimized Code:

```
int a = 5;
int b = 10;
int c = a + b;
cout << c << endl;
```

Input	Output
–	15

12.7.2   Exercises

Academic Exercises

1. **Common Subexpression Elimination:** Given the expression:

$$a = b \times c + d \times e + b \times c$$

   Identify common subexpressions and rewrite the optimized code.

2. **Dead Code Elimination:** Given the code snippet:

```
x = 10;
y = x + 5;
x = 20;
z = y + 1;
```

   Identify dead code and optimize the code.

3. **Loop Invariant Code Motion:** Consider the loop:

```
for (i = 0; i < n; i++) {
    x = y + z;
    a[i] = x * i;
}
```

   Identify loop-invariant computations and move them outside the loop.

Programming Exercises

Problem 1: Constant Folding

Description: Write a program to perform constant folding on arithmetic expressions.

Problem 2: Dead Code Detection

Description: Write a program that detects dead code in simple variable assignments.

Problem 3: Copy Propagation

Description: Implement copy propagation to replace variables by their known copies.

Explore the codes in GitHub
Click or Scan the QR

Chapter 13

Practical Project

Project Overview

This project, named **Compiler Learning Hub**, is an interactive web application designed to help users learn compiler design concepts through practical tools and visualizations. It covers fundamental topics such as grammar analysis, parsing, ambiguity detection, and code generation.

Features

Grammar Analysis

- **FIRST & FOLLOW Sets** - Compute FIRST and FOLLOW sets for context-free grammars.

- **LL(1) Parsing Table** - Generate LL(1) parsing tables with conflict detection.

- **Grammar Parser** - Parse text-based grammar notation into structured format.

Ambiguity & Conflict Resolution

- **Ambiguity Checker** - Detect grammar ambiguity through FIRST set conflicts.

- **Left Recursion Elimination** - Remove direct and indirect left re-

cursion.

- **Left Factoring** - Apply left factoring to eliminate common prefixes.

Language Theory

- **Language** $\rightarrow$ **Regex** - Convert formal language descriptions to regular expressions.

- **Regex** $\rightarrow$ **Language** - Understand what regular expressions represent.

- **Pattern Recognition** - Automatic detection of common language patterns.

Parsing & Code Generation

- **LR(0) Parser** - Generate LR(0) parsing tables and analyze shift-reduce conflicts.

- **Three Address Code** - Convert expressions to TAC representation.

- **Control Flow Graph** - Generate CFGs from TAC with basic block analysis.

Technologies Used

The project is developed using:

- **Backend:** Python with Flask framework.

- **Frontend:** HTML, CSS, and JavaScript for a responsive user interface.

- **Version Control:** Git and GitHub for source code management.

13.1. Repository

The complete source code and documentation are available on GitHub at:

 https://github.com/MNR-Tushar/Compiler-Learning-Hub

 Compiler Learning Hub
Click or Scan the QR

You can clone the repository, explore the code, and contribute to its development.

Live Demo

Explore the live version of the **Compiler Learning Hub** project using the link or QR code below:

 Live Demo

Compiler Learning Hub
compiler-learninghub.onrender.com

Compiler Design Resources

Reference Books

1. **Compilers: Principles, Techniques, and Tools** (Aho, Lam, Sethi, Ullman)

2. **Engineering a Compiler** (Cooper, Torczon)

Compiler Blueprint makes learning compiler design easy and fun. It explains how programs are translated from code to execution with simple examples and clear diagrams. Perfect for students, beginners, and anyone curious about how compilers work.

- Covers both Theory & Lab

- Includes exercises & programs

- Hands-on project included

"Learn by doing, master by understanding."