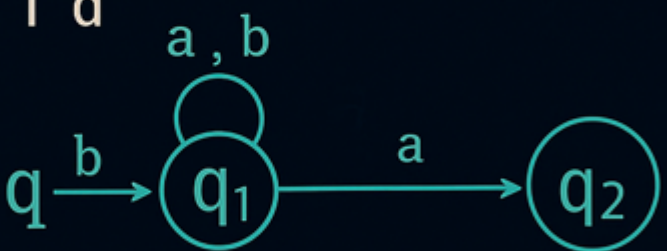
COMPILER DESIGN

A BEGINNER'S GUIDE



Muhammad Razim Al Sami
ABM Nahid Mahbub

Compiler Design: A Beginner's Guide

Muhammad Razim Al Sami
A B M Nahid Mahabub

Department of Computer Science & Engineering
Daffodil International University

2025

Muhammad Razim Al Sami & A B M Nahid Mahabub

Acknowledgements

We would like to express our sincere gratitude to all who have supported and contributed to the development of this book. Special thanks to our mentors, colleagues and family for their encouragement and valuable feedback.

Dedicated to our esteemed course teacher

Musfiqur Rahman

whose guidance and patience have enlightened us throughout this journey.

Special thanks to our loving parents,
whose unwavering support and encouragement have been our greatest strength.

This book is a humble tribute to their faith in us.

Preface

This book aims to provide a clear and beginner-friendly introduction to compiler design. We have structured the content to gradually build your understanding, starting from the fundamentals to more advanced topics. We hope this guide will help students and professionals alike to grasp the essentials of compiler construction.

Contents

List of Figures

List of Tables

Listings

Part I

Foundations of Compiler Design

Chapter 1

Introduction to Compilers

1.1 Overview

A **compiler** is a special software that helps convert programs written by humans into a form that a computer can understand.

When we write code in languages like **C, C++, Java, or Python**, we use high-level instructions that are easy for humans to read and write. However, computers do not understand these high-level languages directly.

The compiler takes this high-level code and translates it into **machine language**, which is made up of binary (0s and 1s) that the computer's processor can execute.

Sometimes, instead of directly converting to machine language, the compiler first changes the code into an **intermediate representation (IR)**. This form is simpler than the original but still not machine code. It helps in optimizing the code and makes it easier to run on different devices.

So, in simple terms, a compiler acts like a **translator** between the programmer and the computer.

The compilation process is not just simple translation; it involves a series of well-defined stages such as:

- **Lexical Analysis** (token generation)
- **Syntax Analysis** (parsing)
- **Semantic Analysis** (meaning checking)
- **Intermediate Code Generation** (IR Translation)
- **Code Optimization** (improving performance)
- **Final Code Generation** (final output code)

Using compilers allows developers to write portable, efficient

and maintainable software across different platforms.

1.2 Historical Context

Understanding how compilers evolved over time helps us to understand why they are so powerful and important in today's software world. The journey of compiler development shows the growth of programming itself — from writing machine code directly to using high-level languages like Python and Java.

Early Days of Compilers (1950s)

- **Before compilers**: In the very early days of computing, programmers had to write programs using binary or assembly language. This was very difficult and time-consuming. There were no tools to convert high-level ideas into machine instructions.
- **A-0 Compiler (1952)**: One of the first steps toward automating programming was the **A-0 System**, developed by **Grace Hopper**. Instead of writing binary instructions directly, programmers could write symbolic names that were linked to machine instructions. This was not a full compiler, but it helped start the journey.
- **FORTRAN Compiler (1957)**: Developed by IBM,

FORTRAN (Formula Translation) was the first high-level programming language. It came with a compiler that could translate mathematical formulas into machine code. FORTRAN made programming easier for scientists and engineers. Its compiler was one of the first to focus on **optimization**, making programs run faster.

1960s–1980s: Growth and Formalization

This was a period when computer scientists started developing formal theories and tools for compiler construction. Programming languages were also growing rapidly.

- **ALGOL and BNF (1960s)**: The **ALGOL** language introduced a clear structure for writing programs. Along with it came the **Backus-Naur Form (BNF)**, a formal way to describe the grammar of programming languages. BNF allowed computer scientists to define syntax rules in a mathematical way. This made it easier to create compilers for new languages.
- **Development of C (1972)**: The **C programming language**, developed at Bell Labs, became very popular. It needed efficient compilers because it was used to build operating systems like UNIX. The success of C led to the development of better and faster compilers.

- **Lex and Yacc (1970s)**: To make compiler construction easier, tools like **Lex** and **Yacc** were created. Lex is used to build lexical analyzers and Yacc helps in building parsers. These tools allowed developers to quickly build front-end parts of compilers by writing rules instead of full code from scratch.

- **Pascal and Ada**: These programming languages introduced structured programming. Compilers for these languages included features like type checking, error detection and modular compilation. They also influenced the design of future languages and compilers.

1990s–2000s: Optimizations and Open Source Compilers

- **Java and the JVM (1995)**: With the release of **Java**, a new model of compilation appeared. Java source code is compiled into **bytecode**, which runs on the **Java Virtual Machine (JVM)**. This allowed Java programs to be platform-independent (write once, run anywhere). The JVM also uses a **Just-In-Time (JIT)** compiler to improve runtime speed.

- **GNU Compiler Collection (GCC)**: GCC became a powerful open-source compiler system that supports

many languages like C, C++, Fortran and more. It became the default compiler for Linux and many open-source projects.

2000s and Beyond: Modern Compiler Technologies

- **LLVM (2003)**: The **Low-Level Virtual Machine (LLVM)** is a modern compiler infrastructure project started at the University of Illinois. It supports modular and reusable compiler components. Many modern languages such as **Rust, Swift, Julia and Clang (C/C++)** use LLVM as their backend. It provides powerful optimizations and supports cross-platform compilation.

- **Just-In-Time (JIT) Compilers**: Languages like **JavaScript (via V8 engine)** and **Python (via PyPy)** use JIT compilation to translate code at runtime for better performance.

- **Multi-language Support and IDE Integration**: Modern compilers are now integrated with development environments (IDEs) like VS Code, IntelliJ and Eclipse. These compilers also provide features like **syntax checking, code completion,** and **error suggestions**.

- **Machine Learning in Compilers**: Some research compilers now use machine learning to improve optimization strategies based on how code performs.

- **WebAssembly**: WebAssembly is a new low-level bytecode format that allows compiled code (from C/C++/Rust) to run in web browsers at near-native speed. This opens new doors for compilers in the world of web development.

Summary

The history of compilers is the story of making programming easier, faster and more reliable. From writing raw machine code to using intelligent compilers that optimize and run code across platforms, the journey of compiler technology has been remarkable. Understanding this history helps us to see how today's compilers are built and how they might continue to evolve.

1.3 Applications of Compilers

Compilers are a core part of modern computing. Without compilers, it would be almost impossible to develop complex software systems. They play a crucial role in various fields by translating high-level code into machine-understandable

instructions.

This section explains the major areas where compilers are used and how they help improve performance, productivity and portability of software.

a. Software Development

In everyday software development, compilers are the most essential tools. They enable programmers to write code in high-level languages and then convert that code into executable machine code that the computer can run.

- For **C and C++**, the most common compiler is **GCC (GNU Compiler Collection)**. It supports optimization, error detection and cross-platform compilation.
- For **Java**, the compiler **javac** converts Java code into bytecode, which is then executed by the **Java Virtual Machine (JVM)**. This makes Java portable across different systems.
- For **Rust**, the compiler **rustc** compiles Rust programs into highly optimized machine code with strong memory safety checks.
- For web development, tools like **TypeScript** have compil-

ers that convert TypeScript code into JavaScript, which can run in any browser.

- IDEs like Visual Studio, Eclipse and JetBrains IntelliJ include built-in compilers that provide features like code suggestion, real-time error checking and debugging.
- Modern compilers help developers with:
  - Error detection during coding
  - Code optimization for performance
  - Platform-independent execution

b. Embedded Systems

Embedded systems are small computer systems that are built into devices like washing machines, smartwatches, industrial robots, or medical instruments. These systems usually have limited memory, processing power and storage.

- Compilers for embedded systems must generate highly efficient code that uses minimal resources.
- **ARM-based microcontrollers** (used in Arduino, Raspberry Pi, etc.) use compilers like **Keil, IAR, GCC ARM** to build firmware.

- Real-time operating systems (RTOS) require fast and predictable code generation, which is handled by the compiler.
- Embedded compilers often include:
  - Memory footprint reduction techniques
  - Code size optimization
  - Power-efficient code generation
- Example: In a pacemaker, the software must be compact, reliable and energy-efficient. The compiler ensures the code is optimized for such critical conditions.

c. Domain-Specific Languages (DSLs)

A **Domain-Specific Language (DSL)** is a programming language designed for a specific set of tasks or a particular application domain. DSLs are not general-purpose but are highly effective within their domain.

- **SQL (Structured Query Language)** is a DSL for managing and querying databases. SQL interpreters and optimizers are part of database compilers.
- **MATLAB** is widely used in engineering and scientific

computation. Its backend has a compiler that optimizes matrix operations.

- **TensorFlow and PyTorch** use graph-based DSLs to define and optimize machine learning models.
- **HTML/CSS preprocessors** like Sass and Less act as DSLs and are compiled into standard CSS.
- Benefits of DSL compilers:
  - Abstract away low-level complexity
  - Generate efficient domain-specific code
  - Validate syntax and optimize performance for a narrow task
- Companies often build their own DSLs for internal tools (e.g., Google's Bazel, Facebook's Hack).

d. Cross-Compilation

Cross-compilation means compiling a program on one machine (host) to run on another machine (target). This is very useful in system development, embedded programming and application distribution.

- For example, a developer may write code on a Windows

computer but compile it to run on an Android phone or Raspberry Pi.

- Cross-compilers are used when the target device does not have enough power or storage to run a full compiler.
- Popular cross-compilers:
  - **arm-none-eabi-gcc** – for ARM Cortex-M microcontrollers
  - **mingw-w64** – for compiling Windows programs from Linux
  - **NDK (Native Development Kit)** – used to compile native code for Android apps
- Advantages of cross-compilation:
  - Faster development using powerful host systems
  - Ability to test on various platforms without rewriting code
  - Build systems like **CMake, Make and Meson** support cross-compilation setups
- Example: IoT devices often require code compiled on a PC and then transferred to the small device due to lack

of in-device resources.

e. Web Development and Frontend Tools

Modern web development also uses compilers, although not always in the traditional sense.

- Tools like **Babel** compile next-generation JavaScript (ES6+) into older JavaScript for browser compatibility.
- Frameworks like **React, Svelte and Vue** have compilers that convert components into optimized JavaScript code.
- **WebAssembly (Wasm)** is a modern compiler target that allows languages like C, C++ and Rust to run in the browser at near-native speed.
- Benefits:
  - Faster page loading and performance
  - Ability to run compiled applications inside browsers
  - Enhanced security and sandboxing

f. Scientific and High-Performance Computing

In scientific fields, compilers help generate highly optimized code to run on supercomputers and clusters.

- **Fortran compilers** are still widely used in physics, weather forecasting and engineering simulations due to their performance.
- **OpenMP, CUDA and OpenCL** compilers are used for parallel and GPU-based computing.
- These compilers generate code that runs on thousands of cores in parallel for maximum speed.
- Compiler optimizations here are critical to ensure:
  - Low-level memory management
  - Vectorization of loops
  - Parallel task distribution

Summary

Compilers are not just tools for translating code — they are central to almost every part of the computing world. Whether it's a desktop application, a mobile app, an embedded system, a scientific simulation, or a web app running in a browser — compilers make it possible to convert human-readable instructions into efficient, optimized machine code.

The more we understand compiler applications, the better we

can appreciate their value in software engineering and system development.

1.4 Tools

Several tools play a crucial role in building modern compilers. These tools help automate different stages of the compilation process — from lexical analysis to code generation and optimization. By using them, compiler developers can focus more on language-specific logic rather than rewriting low-level routines.

- **Lex**:
  - Lex is a lexical analyzer generator.
  - It takes regular expressions as input and generates C code that can tokenize input strings.
  - Lex simplifies the task of scanning input and breaking it down into tokens (identifiers, numbers, keywords, etc.).
  - Lex helps identify invalid tokens early, improving compiler accuracy.
  - Example: If your language has keywords like ‘if‘,

‘else‘, ‘while‘, Lex can be configured to recognize these patterns and generate corresponding token codes.

  - Lex output usually works with Yacc for full syntax parsing.

- **Yacc (Yet Another Compiler Compiler)**:

  - Yacc generates parsers using a context-free grammar.

  - It produces a parser in C, which can handle nested structures and detect syntax errors.

  - Yacc works on the syntax analysis phase, building parse trees or abstract syntax trees (AST).

  - It uses BNF (Backus-Naur Form) like notation to define grammar rules.

  - Example: You can define a rule like ‘expr → expr + expr‘ to parse arithmetic expressions.

  - When integrated with Lex, Yacc can parse full programming languages like C or Pascal.

  - Modern alternatives to Yacc include Bison (GNU

Yacc) and ANTLR (which also supports Java, C#, Python).

- **GCC (GNU Compiler Collection)**:
  - GCC is a production-ready compiler that supports many languages: C, C++, Objective-C, Ada, Fortran, Go, etc.
  - It serves both as a standalone compiler and as a backend for custom compilers.
  - GCC can perform advanced code optimizations like loop unrolling, constant folding, dead code elimination.
  - It supports multiple target architectures (ARM, x86, RISC-V, etc.).
  - GCC includes several stages:
    * Preprocessing (‘cpp‘)
    * Compilation (‘cc1‘)
    * Assembly (‘as‘)
    * Linking (‘ld‘)

  - Developers can write frontends that emit GCC-compatible intermediate code.
  - Many operating systems (Linux, BSD) rely on GCC for kernel and application builds.

- **LLVM (Low Level Virtual Machine)**:
  - LLVM is a modern, modular compiler framework.
  - It uses an intermediate representation (IR) that is language-agnostic and supports multiple optimization passes.
  - It separates frontend, optimizer and backend phases, allowing flexibility and reuse.
  - Clang is an LLVM-based frontend for C/C++/Objective-C that provides faster compilation and better diagnostics.
  - LLVM optimizations include inlining, loop vectorization, SSA-based analysis and target-specific tuning.
  - Languages like Rust, Julia, Swift and Kotlin Native use LLVM for backend compilation.
  - LLVM is also used in GPU programming, virtual

machines and ahead-of-time (AOT) compilation in mobile and game engines.

- LLVM tools include:
  - `llc` – Converts LLVM IR to native machine code.
  - `opt` – Applies optimization passes.
  - `clang-format` – Code formatting tool used in IDEs.
- LLVM's clean API makes it ideal for compiler researchers and academic projects.

Together, these tools reduce the complexity of compiler construction by providing powerful, reusable components for each phase of the compilation pipeline. Developers no longer need to build everything from scratch — instead, they can plug these tools together to create robust, scalable and efficient compilers for modern software systems.

1.5 Architecture

The internal structure of a compiler is generally divided into **three major components**: the front-end, middle-end and

back-end. Each component has a specific role in the compilation pipeline.

a. Front-End

The front-end is responsible for analyzing the source code and converting it into an intermediate format. It performs:

- **Lexical Analysis**: Converts raw source code into tokens.

  Example: From `int x = 10;` it produces tokens like `int`, `x`, `=`, `10`, `;`.

- **Syntax Analysis (Parsing)**: Verifies that the code follows the correct grammar using techniques like LL and LR parsing. It generates a **parse tree** or **abstract syntax tree (AST)**.

- **Semantic Analysis**: Ensures that the parsed code makes logical sense (e.g., variable declarations, type compatibility, scope rules).

If errors are found, the front-end reports them to the developer.

b. Middle-End

This stage takes the intermediate representation from the front-end and **optimizes** it for better performance. Optimizations

can be:

- **Constant Folding**: Precomputing constant expressions
- **Dead Code Elimination**: Removing code that is never used
- **Loop Unrolling**: Improving loop performance
- **Inline Expansion**: Replacing function calls with actual code

The output is still in IR form but more efficient.

c. Back-End

The back-end takes the optimized IR and generates **target-specific machine code**. It handles:

- **Register Allocation**: Assigning variables to physical CPU registers
- **Instruction Selection**: Choosing the best machine instructions
- **Code Emission**: Generating the final binary or assembly code

The output is executable code that can run on the desired

hardware platform.

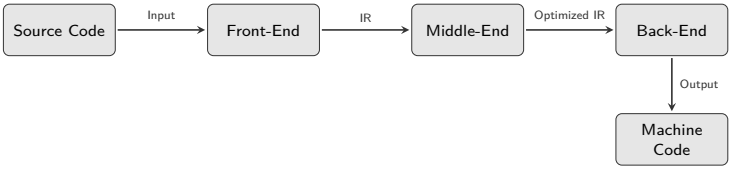


Figure 1.1: Compiler pipeline transformation stages

Chapter 2

Interpreters vs. Compilers

2.1 Introduction

In programming, when we write code in languages like C, Java, or Python, computers cannot understand that code directly. We need a translator that converts our human-readable code into something that a computer can execute. These translators are called **compilers** and **interpreters**.

Both serve the same purpose: to help the computer run our code. However, they do it in different ways. Understanding how they work and differ is a very important concept in compiler design.

2.2 Definition of Compiler

A **compiler** is a special program that translates the entire source code (program) into machine code (or intermediate code) before the program is executed.

Example: When you write a C program and use the `gcc` command to compile it, the compiler checks your whole code, reports errors (if any) and then generates an executable file like `a.exe` or `main.out`.

Key Characteristics:

- Translates entire code at once
- Gives errors only after complete analysis
- Creates an independent executable file
- Faster execution after compilation

2.3 Definition of Interpreter

An **interpreter** is a program that translates and executes the source code line-by-line, one instruction at a time. It does not generate a separate executable file.

Example: When you run a Python script with the `python` command, the interpreter reads the first line, executes it, then reads the next and so on.

Key Characteristics:

- Translates and runs the program step by step
- Stops immediately if it finds an error
- No separate executable file is generated
- Useful for debugging and quick testing

Note

Compilers translate the whole program at once and create an executable, while interpreters execute code line-by-line without generating an executable. Knowing the difference helps choose the right tool for your needs.

2.4 Everyday Analogy

Let's understand the difference with a real-world example:

- **Compiler:** Imagine writing a book and translating the entire book into another language before giving it to someone. That's like compiling – full translation before execution.
- **Interpreter:** Imagine reading a book to someone and translating each sentence aloud as you go. That's like interpreting – line-by-line translation while reading.

2.5 Detailed Comparison Table

Table 2.1: Comparison Between Compiler and Interpreter

Feature	**Compiler**	**Interpreter**
Translation Time	Entire program is translated before execution	Translates line-by-line during execution
Execution Speed	Faster (once compiled)	Slower due to continuous translation
Error Detection	All errors are shown after compiling the whole code	Errors are shown immediately when that line runs
Output File	Creates a standalone executable (.exe or .out)	No file is created, runs directly from source
Memory Usage	Generally efficient after compilation	May use more memory due to overhead
Examples	C, C++, Java (bytecode compiled), Rust	Python, JavaScript, Ruby, MATLAB
Best Use Case	Production-level software, speed-critical apps	Education, testing, scripting, web apps

2.6 Diagram: How They Work

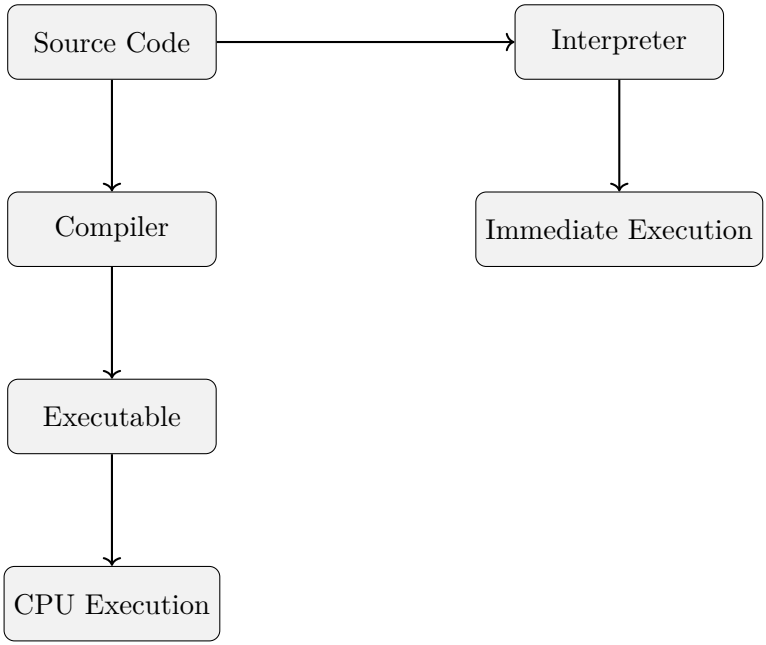


Figure 2.1: Compiler vs. Interpreter Workflow (Black and White)

Figure 2.1 shows how compilers and interpreters work differently to run a program.

- On the left side, the compiler takes the full source code, converts it into an executable file and then the CPU runs that file. This process is done once and the program runs faster.

- On the right side, the interpreter reads the source code line by line and executes it immediately. It doesn't create any separate file.

In short, compilers translate first and run later, while interpreters translate and run at the same time.

2.7 Code Example

Sample Python Program

Listing 2.1: Sample Python code (Interpreter)

```
print("Start")
x = 10
y = 0
print("Result:", x / y)
print("End")
```

Interpreter Behavior:

- It prints "Start"
- Executes `x = 10`, `y = 0`
- Crashes on division by zero
- Never reaches `print("End")`

Sample C Program

Listing 2.2: Sample C code (Compiler)

```c
#include <stdio.h>

int main() {
    printf("Start\n");
    int x = 10, y = 0;
    printf("Result: %d\n", x / y);
    printf("End\n");
    return 0;
}
```

Compiler Behavior:

- Code is compiled first
- Errors (like divide by zero) may not be caught at compile-time
- If compiled, executable will crash during execution

2.8 Advantages and Disadvantages

Category	Compiler	Interpreter
Advantages	• Fast execution after compilation • Creates a standalone executable	• Easy to debug (errors shown immediately) • Good for scripting and quick testing
Disadvantages	• Compilation can be slow • Harder to debug (errors shown after full analysis)	• Slower execution • Needs interpreter every time to run

Table 2.2: Comparison of Compiler and Interpreter

2.9 Modern Hybrid Approaches

Modern programming environments combine the best of both

- **Java:** Uses a compiler to convert source to bytecode and then a JVM interpreter/JIT compiles it again during execution.
- **Python:** Python files are interpreted but also compiled to `.pyc` bytecode behind the scenes.
- **JavaScript:** Modern browsers use JIT compilers (e.g., V8 engine in Chrome) to make JavaScript execution faster.

Summary Table

Compiler	Interpreter
Translates entire program	Translates one line at a time
Produces executable	No executable generated
Runs faster after compilation	Slower overall execution
Best for performance	Best for testing/debugging

Table 2.3: Quick Summary

2.10 Conclusion

Compilers and interpreters are both essential tools in the world of programming. They serve the same goal — helping us run our code — but they do so in very different ways. Whether we choose one or the other depends on our needs: speed and efficiency, or flexibility and ease of debugging. Understanding these tools makes us better programmers and helps us choose the right language or tool for the job.

2.11 Code Examples

Problem 1: Counting String Length Without Using Any Function

Code:

Listing 2.3: Counting string length without using any function

```c
#include <stdio.h>

int main() {
    char str[10000];
    int length = 0;

    printf("Enter a string:\n");
    scanf("%[^\n]", str); // Read input including spaces
         until newline

    while (str[length] != '\0') {
        length++;
    }

    printf("The length of the string is: %d\n", length);
    return 0;
}
```

Sample Input:

```
Hello, this is a sample string!
```

Sample Output:

```
The length of the string is: 31
```

Problem 2: Counting White Spaces in a String

Code:

Listing 2.4: Counting white spaces in a string

```c
#include <stdio.h>

int main() {
    char str[10000];
    int count = 0;
    int index = 0;

    printf("Enter a string:\n");
    scanf("%[^\n]", str);

    while (str[index] != '\0') {
        if (str[index] == ' ')
            count++;
        index++;
    }

    printf("The number of white spaces is: %d\n", count)
        ;
    return 0;
}
```

Sample Input:

```
Count the number of spaces in this sentence.
```

Sample Output:

```
The number of white spaces is: 7
```

Problem 3: Removing White Spaces from a String

Code:

Listing 2.5: Removing white spaces from a string

```
#include <stdio.h>

int main() {
    char str[10000];

    printf("Enter a string:\n");
    scanf("%[^\n]", str);

    for (int i = 0; str[i] != '\0'; ++i) {
        if (str[i] == ' ') {
            for (int j = i; str[j] != '\0'; ++j) {
                str[j] = str[j + 1];
            }
            i--; // Check this index again after shifting
        }
    }

```

```
18      printf("After removing white space: %s\n", str);
19      return 0;
20  }
```

Sample Input:

```
Remove all spaces from this string.
```

Sample Output:

```
After removing white space: Removeallspacesfromthisstring.
```

2.12 Viva/Exam-Style Questions

Basic Questions:

1. What is the fundamental difference between an interpreter and a compiler?
2. How does an interpreter execute code compared to a compiler?
3. Which one is faster in execution—compiler or interpreter? Why?
4. Does an interpreter produce an executable file? Why or why not?

5. What is the role of a compiler in program execution?

Intermediate Questions:

6. Explain the steps involved in compilation. How does this differ from interpretation?

7. Can a language be both compiled and interpreted? Give an example.

8. Why do interpreted languages generally offer better debugging capabilities?

9. What is Just-In-Time (JIT) compilation? How does it bridge the gap between interpreters and compilers?

10. Why are scripting languages like Python and JavaScript often interpreted rather than compiled?

Advanced Questions:

11. How does memory management differ between compiled and interpreted programs?

12. What are the advantages of using an interpreter during the development phase?

13. Can a compiler generate intermediate code like bytecode?

How does this affect execution?

14. Why are security vulnerabilities sometimes easier to exploit in interpreted languages?

15. Compare the portability of compiled vs. interpreted programs.

Conceptual & Scenario-Based Questions:

16. If a program has a syntax error, how would the behavior differ between a compiler and an interpreter?

17. Would you prefer a compiler or an interpreter for a real-time system? Justify your answer.

18. How does an interpreter handle dynamic typing compared to a compiler?

19. Explain the trade-offs between compilation and interpretation in terms of speed and flexibility.

20. Can you convert an interpreter into a compiler? What changes would be needed?

Chapter 3

Phases of a Compiler

3.1 Introduction

High-level languages like C, Java, and Python are human-readable but not directly understandable by a computer's processor. A **compiler** acts as a translator, converting this code into **machine code**.

However, this conversion is not done in a single step. The process is broken into well-defined stages called the **phases of a compiler**. Each phase performs a specific function and passes its output to the next phase. This modular design makes compilers easier to build, debug, and maintain.

3.2 Major Phases of a Compiler

Phase Classification

Compilers are usually structured into three major components:

- **Front-End**: Analyzes source code and builds an intermediate form (Lexical, Syntax, Semantic Analysis).
- **Middle-End**: Optimizes the intermediate representation (Optimization).
- **Back-End**: Generates target machine code (Code Generation).

3.3 Detailed Explanation of Phases

1. Lexical Analysis (Scanner)

The compiler reads raw source code character-by-character and groups sequences into **tokens**, such as keywords, identifiers, operators, etc.

- Removes comments and white spaces.
- Detects lexical errors (e.g., invalid identifiers).

Example: For the line `int sum = a + b;`, the output tokens are:

`[int], [sum], [=], [a], [+], [b], [;]`

2. Syntax Analysis (Parser)

This phase checks the grammatical structure of the token stream using grammar rules. It builds a **parse tree** or **abstract syntax tree (AST)**.

- Detects syntax errors (e.g., missing semicolons, unmatched braces).
- Ensures constructs follow correct syntax.

Example: For `int sum = a + b;`, it checks:

- Is `int` a valid type?
- Does the expression `a + b` follow operator precedence?

3. Semantic Analysis

Ensures the program is logically correct beyond just syntax. It verifies:

- Variables are declared before use.
- Types are compatible in expressions (e.g., int + float).
- No scope violations or redefinitions.

Example: `a = "Hello" + 5;` is semantically incorrect (incompatible types).

4. Intermediate Code Generation (IR)

Produces an abstract form of code between high-level and machine code, making it easier to analyze and optimize.

- Common formats: **Three-Address Code (TAC)**, **Control Flow Graphs**.
- Facilitates portability across architectures.

Example (TAC):

```
t1 = a + b
sum = t1
```

5. Code Optimization

Improves the intermediate code by reducing resource usage and increasing performance.

- **Constant Folding**: Compute constant expressions at compile-time.
- **Dead Code Elimination**: Remove unreachable or unused code.

- **Loop Optimization**: Minimize expensive operations inside loops.

Example:

```
a = 2 + 3;      → a = 5;
b = a * 1;      → b = a;
```

6. Final Code Generation

The final phase where the compiler generates the actual machine-level or assembly code that can be executed on hardware.

- Performs register allocation.
- Selects instruction sets for the target CPU.

Example:

```
MOV R1, a
MOV R2, b
ADD R2, R1
MOV sum, R2
```

3.4 Compiler Phase Diagram

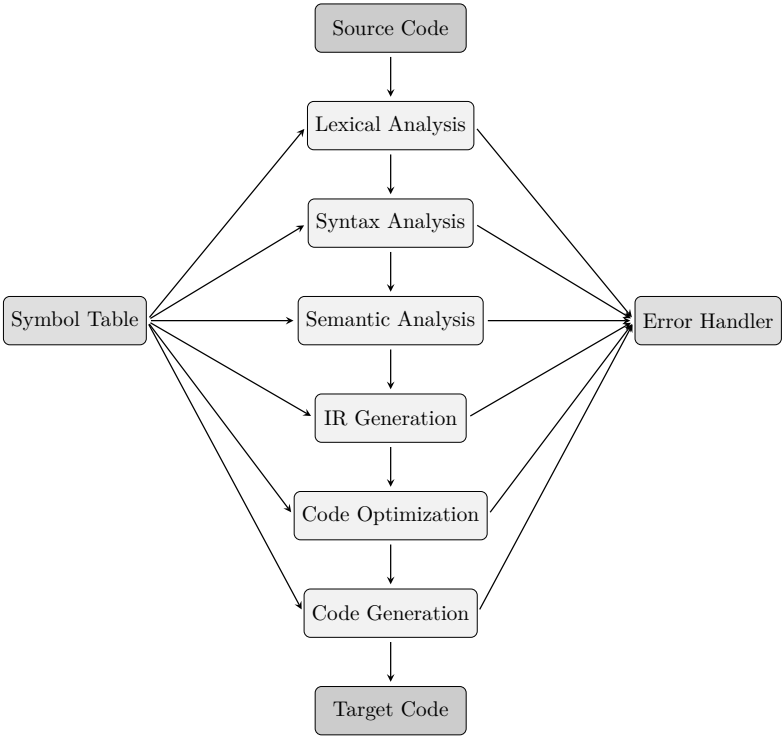


Figure 3.1: Phases of a Compiler (Black and White)

3.5 End-to-End Example: Summation Program

Let us walk through the phases using this simple C code:

Listing 3.1: Simple C program for summation

```c
#include <stdio.h>

int main() {
    int a = 5;
    int b = 10;
    int sum = a + b;
    printf("Sum: %d\n", sum);
    return 0;
}
```

Phase-wise Breakdown

1. Lexical Analysis

The source is broken into tokens:

`[int], [main], [(], [)], [{], [int], [a], [=], [5], [;], ...`

2. Syntax Analysis

Checks structural rules:

- Is `int a = 5;` a valid declaration?
- Builds a syntax tree:

```
main()
 └── Block
      ├── int a = 5;
      ├── int b = 10;
      ├── int sum = a + b;
      └── printf(...)
```

3. Semantic Analysis

Checks for meaning:

- Are `a`, `b` declared before use?
- Is `sum = a + b;` type-compatible?

4. Intermediate Code (TAC)

```
t1 = 5
t2 = 10
t3 = t1 + t2
sum = t3
call printf("Sum: %d\n", sum)
```

5. Optimization

Since a and b are constants:

```
sum = 15
call printf("Sum: %d\n", 15)
```

6. Final Code Generation

```
MOV R1, #5
MOV R2, #10
ADD R2, R1
MOV Sum, R1
```

3.6 Example of Code

Example 1: Multiple Line Input/Output

```
#include<stdio.h>
#include<string.h>
int main(){
    char str[10000];
    scanf("%[^]s", str);
    printf("%s\n", str);
    return 0;
}
```

Listing 3.2: C Program to handle multi-line input

Input:

Hello World
This is multi-line
input test
% <– Trigger character

Output:

Hello World
This is multi-line
input test

Explanation: This program attempts to read multiple lines of input as a single string and prints it. The ‘scanf("

Example 2: Single-Line Comment Extractor

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

int main() {
    char str[10000];
    scanf("%[^]s", str);

    for(int i = 0; str[i] != '\0'; i++) {
        if(str[i] == '/' && str[i+1] == '/') {
            while(str[i] != '\n' && str[i] != '\0') {
                printf("%c", str[i]);
                i++;
            }
        printf("\n");
        }
```

```
    }
    return 0;
}
```

Listing 3.3: C Program to extract single-line comments

Input:

```
// This is a comment
int x = 5; // Initialize x
/* Multi-line comment */
print("Hello"); // Output
%
```

Output:

```
// This is a comment
// Initialize x
// Output
```

Explanation: This program reads an entire block of input and extracts single-line comments (those starting with '//'). It scans through the string, and whenever it finds the '//' pattern, it prints the text from there until the end of the line. Multi-line comments like '/* ... */' are ignored. This is useful for identifying and extracting inline or full-line comments in C/C++ code.

Example 3: Line Counter

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

```c
int main(){
    char str;
    int count = 0;
    while((str = getchar()) != EOF)
    {
        if(str == '\n'){
            count++;
        }
    }
    printf("%d", count);
    return 0;
}
```

Listing 3.4: C Program to count lines

Input:

Line 1
Line 2
Line 3
Line 4
Ctrl+D (EOF)

Output:

4

Explanation: This program reads user input one character at a time until EOF (‘Ctrl+D‘ on Unix/Linux or ‘Ctrl+Z‘ on Windows). It increments a counter each time it encounters a newline character

n, effectively counting how many lines the user entered. This is a common utility in text-processing programs and compilers to track line numbers accurately.

Practice Exercises

Exercise Solve the following problems to deepen your understanding of multi-line input handling in C:

- **Exercise 1: Input until "END"** Modify the program to take input until the user types the word `"END"` (case-sensitive) on a new line. After detecting "END", the program should stop reading and print the complete input except the line containing "END".
- **Exercise 2: Lines Starting with Capital Letter** Write a program that reads multiple lines of input and prints only those lines that begin with a capital letter (A–Z).
- **Exercise 3: Word Count Until '%'** Write a program to count how many words are in a multi-line input. The input should be terminated by the ‘
- **Exercise 4: Reverse Each Line** Write a program that reads multi-line input and prints each line in reverse order, while keeping the original line sequence intact.

The input should be terminated using ‘

Table 3.1: Identifiers Table

Lexeme	Token
x	<id,1>
p	<id,2>
q	<id,3>
y	<id,4>

Table 3.2: Operators Table

Lexeme	Token
=	<op,1>
+	<op,2>
-	<op,3>
/	<op,4>
*	<op,5>

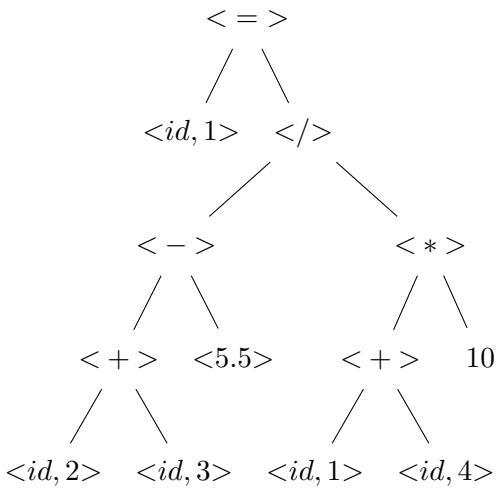
Table 3.3: Constants Table

Lexeme	Token
5.5	<c,1>
10	<c,2>

The tokenized form of the equation:

$\langle id, 1\rangle\ \langle =\rangle\ \langle id, 2\rangle\ \langle +\rangle\ \langle id, 3\rangle\ \langle -\rangle\langle c, 1\rangle\ \langle /\rangle\ \langle id, 1\rangle\ \langle +\rangle\ \langle id, 4\rangle\langle *\rangle\ \langle c, 2\rangle$

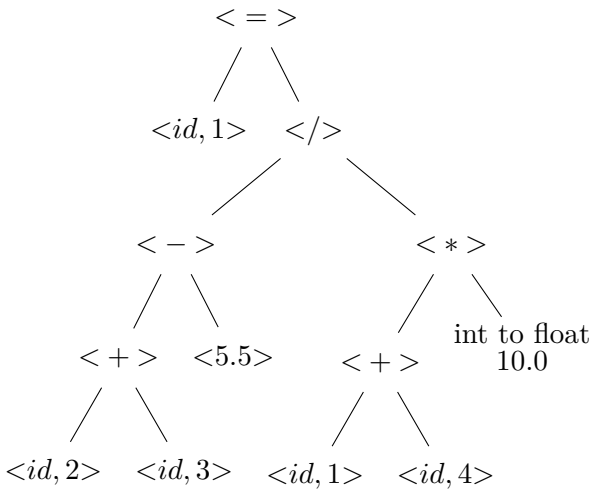
Syntax Tree



Note

A syntax tree, also called a parse tree, represents the hierarchical syntactic structure of source code according to grammar rules. Internal nodes denote operators or grammar constructs, while leaf nodes represent operands like identifiers or literals. Syntax trees are essential in compilers for semantic analysis, optimization, and error detection.

Semantic Tree



Note

A semantic tree represents the meaning of a program's constructs, focusing on the operations and their evaluation rather than just the syntax. It often includes additional information such as data types, conversions, and computed values. Semantic trees are used during compilation for type checking, code optimization, and generating intermediate code.

Part II

Formal Languages and Automata

Chapter 4

Context-Free Grammars

4.1 Introduction

In the world of compiler design, understanding the structure of programming languages is important. This is where Context-Free Grammars (CFGs) come in. CFGs are used to define the syntax of programming languages. They are powerful enough to describe most of the syntactic constructs in modern programming languages.

4.2 What is a Grammar?

A grammar is a set of rules that define how strings in a language can be formed. In computer science, grammars are used to describe how programming statements are constructed.

Example: Consider a simple expression like:

```
x = 5 + y
```

A grammar would describe how to write such expressions using variables, numbers, and operators.

4.3 Formal Definition of a Context-Free Grammar

A Context-Free Grammar (CFG) is defined as a 4-tuple:

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

Where:

- V = A finite set of variables (non-terminal symbols)
- Σ = A finite set of terminal symbols (actual characters/symbols)

- P = A finite set of production rules (how symbols can be replaced)
- S = A special start symbol (one of the non-terminals)

Components Explained

1. Terminals (Σ)

Terminals are the **basic symbols** that form the strings of the language. These cannot be further expanded and appear as-is in the generated strings. **Examples:**

- Letters: `a`, `b`, `c`
- Digits: `0`, `1`, `2`
- Operators: `+`, `*`, `=`
- Punctuation: `(`, `)`, `;`

2. Non-Terminals (V)

Non-terminals are **abstract symbols** that represent sets of strings. They are expanded using production rules until only terminals remain. **Conventions:**

- Capitalized names: E (Expression), T (Term), F (Factor)

- The start symbol is a designated non-terminal.

3. Production Rules (P)

Productions define how non-terminals can be rewritten as sequences of terminals and/or non-terminals.
Examples:

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to \texttt{id} \mid \texttt{num}$$

4. Start Symbol (S)

The start symbol is a **specific non-terminal** ($S \in V$) from which all derivations begin. It represents the entire language defined by the grammar. **Note:** By convention, the start symbol is the left-hand side of the first production rule.

Visualization

Component	**Example**
Terminals	`+, *, id, 42`
Non-Terminals	*E, T, F*
Production	$E \to E + T$
Start Symbol	*E*

Formal Definition of a Context-Free Grammar

A **context-free grammar (CFG)** is defined as a 4-tuple $G = (V, \Sigma, R, S)$ where:

- V is a finite set of **non-terminal** symbols (variables)
- Σ is a finite set of **terminal** symbols ($V \cap \Sigma = \emptyset$)
- R is a finite set of **production rules** of form:

$$A \to \alpha \quad \text{where } A \in V, \alpha \in (V \cup \Sigma)^*$$

- $S \in V$ is the **start symbol**

Key Properties:

- Rules are "context-free" because $A \to \alpha$ applies regardless of surrounding symbols
- $(V \cup \Sigma)^*$ denotes all possible strings over V and Σ (including ϵ)
- The grammar generates a context-free language

Example Grammar

Consider the following unambiguous grammar for arithmetic expressions with assignments:

$$
\begin{aligned}
S &\rightarrow \mathit{result} = E \\
E &\rightarrow E + T \mid T \\
T &\rightarrow T * F \mid F \\
F &\rightarrow \mathit{digit} \mid (E) \\
\mathit{digit} &\rightarrow 0 \mid 1 \mid 2 \mid 3 \mid 4 \mid 5 \mid 6 \mid 7 \mid 8 \mid 9
\end{aligned}
$$

This grammar generates assignments like:

$$\texttt{result} = 5 * 3 + (2 + 7)$$

Parse Tree for `result = 5 * 3 + (2 + 7)`

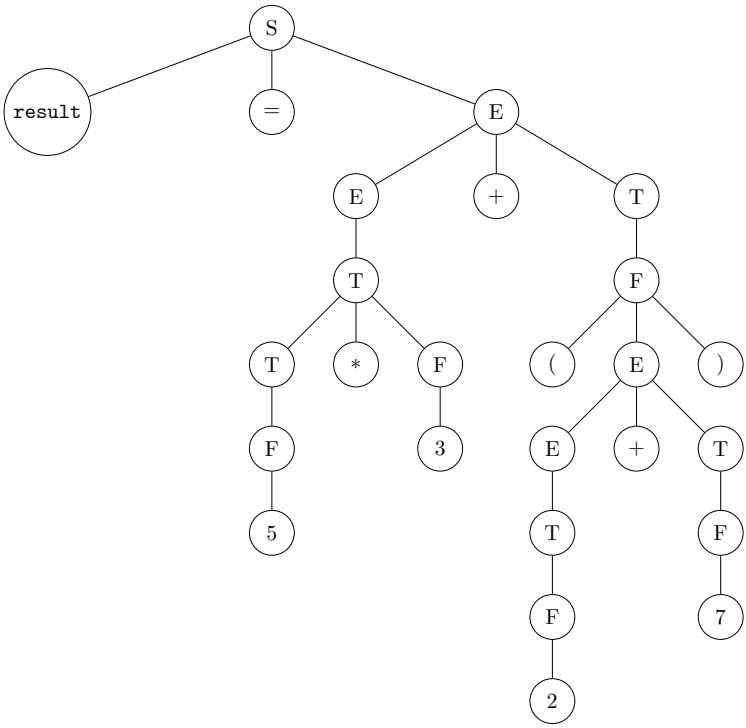


Figure 4.1: Parse tree for `result = 5 * 3 + (2 + 7)`

4.4 Derivations

Derivation is the systematic process of applying production rules in a Context-Free Grammar (CFG) to generate strings belonging to the language. Given a grammar $G = (V, \Sigma, P, S)$,

a derivation starts with the start symbol S and repeatedly replaces non-terminals according to the productions in P until only terminals remain.
In short, derivation is the process of applying production rules to get strings.

4.4.1 Types of Derivations

Leftmost Derivation

In a **leftmost derivation**, at each step, the *leftmost non-terminal* in the current sentential form is always expanded first. This approach ensures a predictable left-to-right expansion order.

Example: Consider the grammar:

$$E \to E + E \mid E * E \mid F$$
$$F \to (E) \mid \text{id} \mid \varepsilon$$

A leftmost derivation for $\mathrm{id} + \mathrm{id} * \mathrm{id}$:

$$\begin{aligned}
E &\Rightarrow E + E \quad (\text{Choose } E \to E + E) \\
&\Rightarrow F + E \quad (\text{Leftmost } E \to F) \\
&\Rightarrow \mathrm{id} + E \quad (\ F \to \mathrm{id}) \\
&\Rightarrow \mathrm{id} + E * E \quad (\ E \to E * E) \\
&\Rightarrow \mathrm{id} + F * E \quad (\text{Leftmost } E \to F) \\
&\Rightarrow \mathrm{id} + \mathrm{id} * E \quad (\ F \to \mathrm{id}) \\
&\Rightarrow \mathrm{id} + \mathrm{id} * F \quad (\ E \to F) \\
&\Rightarrow \mathrm{id} + \mathrm{id} * \mathrm{id} \quad (\ F \to \mathrm{id})
\end{aligned}$$

Rightmost Derivation

In a **rightmost derivation** (also called *canonical derivation*), the *rightmost non-terminal* is always expanded first. This mirrors how a bottom-up parser (e.g., LR parser) would reduce the string.

Example: Using the same grammar, a rightmost derivation

for $\mathrm{id} + \mathrm{id} * \mathrm{id}$:

$$
\begin{aligned}
E &\Rightarrow E * E \quad (\text{Choose } E \to E * E) \\
&\Rightarrow E * F \quad (\text{Rightmost } E \to F) \\
&\Rightarrow E * \mathrm{id} \quad (\ F \to \mathrm{id}) \\
&\Rightarrow E + E * \mathrm{id} \quad (\ E \to E + E) \\
&\Rightarrow E + F * \mathrm{id} \quad (\text{Rightmost } E \to F) \\
&\Rightarrow E + \mathrm{id} * \mathrm{id} \quad (\ F \to \mathrm{id}) \\
&\Rightarrow F + \mathrm{id} * \mathrm{id} \quad (\ E \to F) \\
&\Rightarrow \mathrm{id} + \mathrm{id} * \mathrm{id} \quad (\ F \to \mathrm{id})
\end{aligned}
$$

4.5 Ambiguity in Grammars

A grammar is **ambiguous** if there exists at least one string in the language that can generate multiple distinct parse trees. This typically occurs when:

- The grammar allows multiple operator precedence interpretations
- The grammar has nested productions without clear grouping
- There are conflicting derivation paths for the same input

Example:

$$E \to E + E \mid E * E \mid id$$

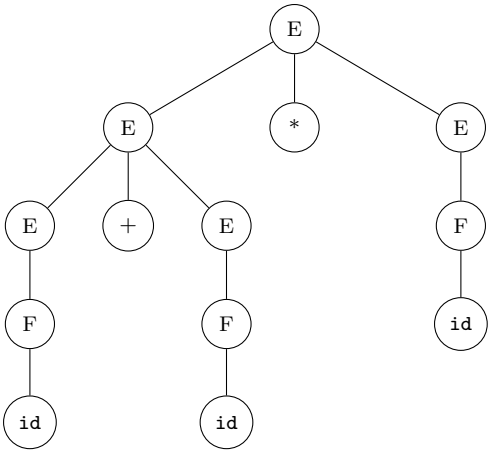
The string `id + id * id` can be parsed in two different ways:

- (id + id) * id
- id + (id * id)

This makes the grammar ambiguous, because it has multiple different parse trees.

Parse Trees:

Addition First



$(id + id) * id$

Multiplication First

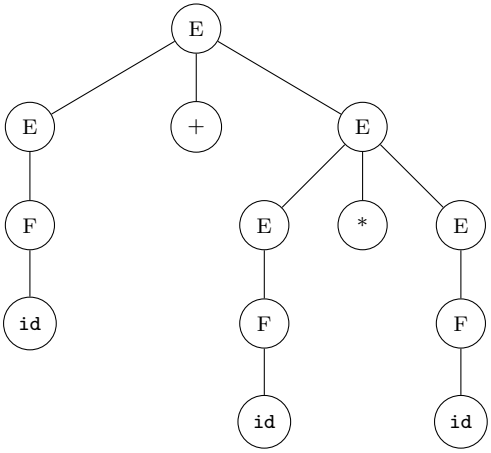


Figure 4.2: Two interpretations of `id + id * id`

Resolving Ambiguity

The following ambiguous grammar allows multiple parse trees for the same expression due to a lack of precedence and associativity rules:

$$E \to E + E \mid E * E \mid id$$

Problem: This grammar does not define whether addition or multiplication should be performed first. For example, the expression `id + id * id` can be parsed in two different ways:

- As `(id + id) * id` — Addition first
- As `id + (id * id)` — Multiplication first

Solution: To remove ambiguity, we can:

1. **Rewrite the grammar** to reflect operator precedence (multiplication over addition) and left associativity:

$$E \to E + T \mid T$$
$$T \to T * F \mid F$$
$$F \to (E) \mid id$$

This grammar ensures:

- `*` has higher precedence than `+`
- Both `+` and `*` are left-associative

2. **Use parentheses** explicitly in expressions to enforce the intended grouping. For example:
   - `(id + id) * id`
   - `id + (id * id)`
3. **Define operator precedence and associativity rules** separately and have the parser apply them when building the parse tree, even if the grammar itself is ambiguous.

4.6 Viva/Exam Questions

1. Define a context-free grammar.
2. What is a parse tree?
3. Differentiate between leftmost and rightmost derivation.
4. What is ambiguity in a grammar?
5. How do we remove left recursion?

4.7 Conclusion

Context-Free Grammars are the foundation of programming language syntax. They help us define how valid statements are formed, and how a compiler can check the correctness of source code. A solid understanding of CFGs makes it easier to grasp how parsers and compilers work in real-world systems.

Chapter 5

Regular Expressions

5.1 Introduction

In the previous chapter, we discussed Lexical Analysis and how it breaks down the source code into meaningful tokens. However, the Lexical Analyzer must have a systematic way of recognizing and classifying tokens efficiently. This is where Regular Expressions (REs) come in.

Regular expressions provide a concise and precise way to define patterns for recognizing keywords, identifiers, numbers, and other syntactic structures. In this chapter, we will explore how regular expressions are used to define tokens in a programming

language, study their operators, and understand language conversions using RE.

Key Insight: Regular expressions form the theoretical foundation for lexical analysis in compilers. They allow precise specification of token patterns that can be efficiently implemented using finite automata.

5.2 Understanding Regular Expressions

A **Regular Expression (RE)** is a symbolic representation of a pattern used to match sequences of characters in text. It is especially useful in defining token structures in programming languages.

5.2.1 Operators of Regular Expressions

- **Literals:** Match specific characters. E.g., `a` matches "a". These are the basic building blocks.
- **Concatenation:** Sequence match. E.g., `ab` matches "ab". Represents ordering of characters.
- **Alternation (|):** Choice. E.g., `a|b` matches "a" or "b". Similar to logical OR.
- **Kleene Star (*):** Zero or more repetitions. `a*` matches

"", "a", "aa", etc. Crucial for variable-length patterns.

- **Kleene Plus (+):** One or more repetitions. `a+` matches "a", "aa", etc. Ensures at least one occurrence.

- **Grouping ( ):** Grouping subexpressions. E.g., `(ab)+` matches "ab", "abab", etc. Controls operator precedence.

Note: The empty string is represented by ε. This is fundamental for patterns that can be absent.

5.2.2 Extended Operators

While not part of formal regular expressions, these are commonly used:

- `[a-z]`: Character ranges (matches any lowercase letter)

- `.`: Wildcard (matches any single character)

- `^`: Start of string

- `$`: End of string

5.3 Regular Expressions for Token Recognition

5.3.1 Identifiers and Keywords

Identifiers start with a letter or underscore and may contain letters, digits, or underscores:

```
[A-Za-z_][A-Za-z0-9_]*
```

Explanation: - `[A-Za-z_]` matches first character (letter or underscore) - `[A-Za-z0-9_]*` matches zero or more subsequent characters (letters, digits, underscores)

5.3.2 Integer and Floating-Point Numbers

- **Integers:** `[0-9]+`
  Matches sequences of one or more digits: 0, 123, 987654

- **Floating-point:** `[0-9]+\.[0-9]+`
  Matches: 3.14, 0.5, 123.456
  Note: The backslash escapes the dot (.), which is a special character

5.3.3 Operators and Punctuation

- **Arithmetic:** `\+ | \- | \* | \/`
  Note: Special characters must be escaped
- **Assignment:** `= | \+= | \-= | \*=`
  Compound operators like +=, -=, etc.
- **Punctuation:** `[;,{}`

  `]`
  Matches single characters: ; , [ ] ( )

5.4 Language-to-Regular Expression Conversion

Regular expressions define languages (sets of strings). Here are common conversions:

1. **All strings of four letters:** `[A-Za-z][A-Za-z][A-Za-z][A-Za-z]`
   Example: "word", "test", "abcd"
2. **Strings starting with a letter:** `[A-Za-z]([A-Za-z0-9])*`
   Example: "a", "var1", "tempValue"
3. **At least one `ab`:** `(ab)+`

Example: "ab", "abab", "ababab" (but not "a" or "b")

4. **Even number of `aa`:** `(aa)*`
   Example: "", "aa", "aaaa" (but not "a" or "aaa")

5. **Odd number of `bb`:** `b(bb)*`
   Example: "b", "bbb", "bbbbb" (but not "" or "bb")

6. **Start with `aa`:** `(aa)(a|b)*`
   Example: "aa", "aab", "aaba" (but not "baa")

7. **Start with `aa`, end with `bb`:** `(aa)(a|b)*(bb)`
   Example: "aabb", "aababb", "aaabbb"

8. **No substring `abb`:** `b*(a(\varepsilon|b))*`
   Ensures no "abb" appears anywhere

9. **At most two `bs`:** `a*(\varepsilon|b)a*(\varepsilon|b)a*`
   Example: "aa", "aba", "baa" (but not "bbba")

5.5 Regular Expression to Language Conversion

Understanding what strings an RE matches:

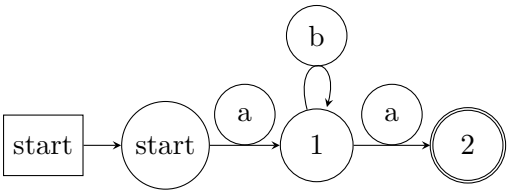
- `a | b` → {a, b} (exactly one 'a' or one 'b')

- `(a|b)(a|b)` → {aa, ab, ba, bb} (all 2-character combinations)
- `a*` → {ε, a, aa, aaa, ...} (any number of 'a's including none)
- `(a|b)*` → All strings of a's and b's (including empty string)
- `a | a*b` → {"a", "b", "ab", "aab", "aaab", ...} (either single 'a' or one/more 'a's followed by 'b')

5.6 Transition Diagrams

A **Transition Diagram** is a graph-based way to visualize RE behavior. It consists of:

- **States:** Represented as circles
- **Start state:** Indicated by an incoming arrow
- **Final state(s):** Indicated by double circles (accepting states)
- **Transitions:** Arrows labeled with input symbols



Example: This transition diagram implements the pattern `ab*a`. It starts at 'start', reads an 'a', then any number of 'b's (including zero), then another 'a' to reach the accepting state.

5.7 Programming Examples

Example 1: RE - (ab)^2+

Pattern: Strings with at least two repetitions of "ab"
Valid Examples: `abab`, `abababab`, `ababababab`
Invalid Examples: "ab", "aabb", "abba"

```c
#include <stdio.h>
#include <string.h>
int isValid(char str[]) {
    int len = strlen(str);
    if (len < 4 || len % 2 != 0) return 0;
    for (int i = 0; i < len; i += 2) {
        if (str[i] != 'a' || str[i+1] != 'b') return 0;
    }
    return 1;
}
```

```c
int main() {
    char str[100];
    printf("Enter a string: ");
    scanf("%s", str);
    if (isValid(str))
        printf("Valid: Matches (ab)^2+\n");
    else
        printf("Invalid: Does not match\n");
    return 0;
}
```

Listing 5.1: Match (ab)^2+ in C

Sample Input/Output:

```
Input: ababab
Output: Valid: Matches (ab)^2+

Input: ababc
Output: Invalid: Does not match
```

Example 2: Validate Email Address

Pattern: Simple email like `name@example.com`

RE: `[a-zA-Z0-9._]+@[a-z]+\.[a-z]+`

Valid Examples: `user@domain.com`, `name123@mail.org`

Invalid Examples: `user@.com`, `@domain.com`, `user@domain`

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

int isEmail(char email[]) {
    int at = 0, dot = 0;
    for (int i = 0; email[i]; i++) {
        if (email[i] == '@') at = i;
        else if (email[i] == '.') dot = i;
    }
    return at > 0 && dot > at + 1 && dot < strlen(email)
     - 1;
}
int main() {
    char email[100];
    printf("Enter email: ");
    scanf("%s", email);
    if (isEmail(email))
        printf("Valid Email\n");
    else
        printf("Invalid Email\n");
    return 0;
}
```

Listing 5.2: Email Validation in C

Sample Input/Output:

```
Input: abc123@gmail.com
Output: Valid Email

Input: invalid.email@
Output: Invalid Email
```

Example 3: Validate `ab*a` Pattern

Pattern: Strings starting with 'a', followed by zero or more 'b's, ending with 'a'

Valid Examples: "aa", "aba", "abba", "abbbba"

Invalid Examples: "ab", "ba", "abc", "a"

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

int isValid(char *str) {
    int len = strlen(str);

    // Must start with 'a' and end with 'a'
    if (len < 2 || str[0] != 'a' || str[len-1] != 'a')
        return 0;

```

```c
    // Middle characters can only be 'b's
    for (int i = 1; i < len-1; i++) {
        if (str[i] != 'b')
            return 0;
    }

    return 1;
}

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

    if (isValid(input))
        printf("Valid: Matches ab*a pattern\n");
    else
        printf("Invalid: Does not match\n");

    return 0;
}
```

Listing 5.3: Validate ab*a pattern in C

Sample Input/Output:

```
Input: abba
Output: Valid: Matches ab*a pattern
```

```
Input: abcba
Output: Invalid: Does not match
```

5.8 Exercises

Conceptual Questions

1. Define a regular expression and explain its three fundamental operations.

2. What is the RE for strings with an even number of `aa` and `bb`? Explain your solution.

3. Write a more comprehensive regular expression for email addresses that includes uppercase domains.

4. Describe the strings matched by `(ab*)+c` with examples.

5. What is the RE for binary strings ending with "1" followed by exactly one more bit?

6. Explain the difference between `a*` and `a+` with examples.

7. Why must special characters like '.' and '*' be escaped in regular expressions?

Programming Tasks

1. Write a program to validate `(abc)*` pattern (e.g., "abc", "abcabc", "")
2. Implement a validator for `b(a|b)*b` pattern (starts/ends with 'b', middle is 'a's/'b's)
3. Create a program for `(0|1)*1(0|1)` (binary strings where second last character is '1')
4. Enhance the email validator to support uppercase letters and domains like .co.uk
5. Write a program to match `(ab)^2+` using finite automaton principles
6. Implement a validator for phone numbers in (XXX) XXX-XXXX format

5.9 Conclusion

In this chapter, we explored the foundational concepts of Regular Expressions and how they contribute to lexical analysis. We examined practical patterns used in token recognition and conversion between RE and language, providing a solid foundation for automata theory.

Key Takeaways: 1. Regular expressions provide a concise way to define token patterns 2. Basic operations (concatenation, alternation, Kleene star) form the core of RE 3. REs can be visually represented using transition diagrams 4. Efficient pattern matching algorithms can be implemented based on RE 5. REs are fundamental to lexical analysis in compiler design

The patterns and techniques covered in this chapter form the basis for building lexical analyzers in compilers. In the next chapter, we'll explore how these regular expressions are implemented using finite automata for efficient token recognition.

Chapter 6

Finite Automata

6.1 Introduction to Finite Automata

Finite automata are abstract machines that model computation with limited memory. They consist of states and transitions that change state based on input symbols. These models are fundamental in compiler design, particularly for lexical analysis (scanning), where they efficiently recognize tokens in source code.

Key characteristics:

- **Finite states**: Limited memory (only current state)

- **Input alphabet**: Fixed set of valid input symbols
- **Determinism vs Nondeterminism**: Single vs multiple possible transitions
- **Acceptance**: Input accepted if ends in accepting state

6.2 Deterministic Finite Automata (DFA)

DFAs are automata where each state-input pair has exactly one transition. Formally, a DFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q: Finite set of states
- Σ: Input alphabet
- $\delta : Q \times \Sigma \to Q$: Transition function
- $q_0 \in Q$: Initial state
- $F \subseteq Q$: Set of accepting states

DFAs process input strings left-to-right, one symbol at a time. The computation always follows a single path, making execution efficient ($O(n)$ time for input length n).

6.2.1 Example 1: DFA for Binary Strings Ending with '01'

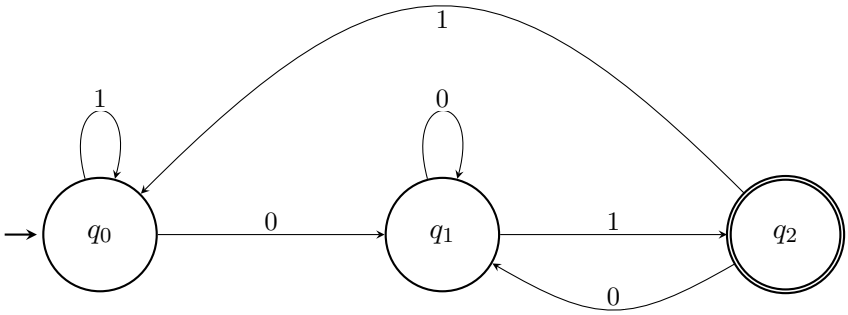


Figure 6.1: DFA for binary strings ending with '01'

State Functions:

- q_0: No meaningful suffix
- q_1: Last symbol was 0
- q_2: Last two symbols were 01 (accepting)

Execution Trace for "1101":

1. Start: q_0
2. '1': $q_0 \to q_0$ (loop)
3. '1': $q_0 \to q_0$ (loop)

4. '0': $q_0 \to q_1$

5. '1': $q_1 \to q_2$ (accept)

6.2.2 Example 2: DFA for Even Number of 0s and Odd Number of 1s

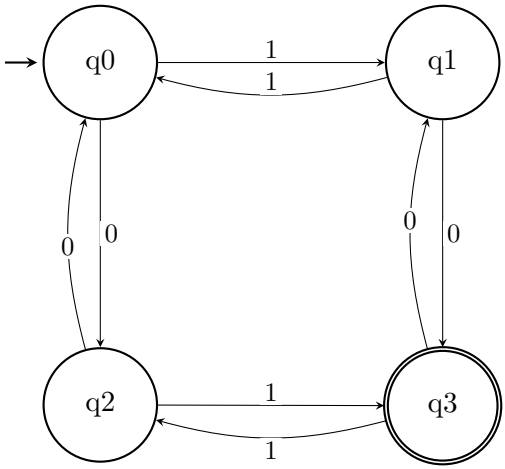


Figure 6.2: DFA for even number of 0s and odd number of 1s.

State Meanings:

- q0 = Even 0s, Even 1s (initial)
- q1 = Even 0s, Odd 1s
- q2 = Odd 0s, Even 1s

- q3 = Odd 0s, Odd 1s (accepting)

Key Insight: This DFA tracks parity of 0s and 1s separately. Each state represents a combination of the two parities.

6.3 Nondeterministic Finite Automata (NFA)

NFAs generalize DFAs by allowing:

- Multiple transitions for the same state-input pair
- ϵ-transitions (state changes without input)
- No transition for some state-input pairs

Formally, an NFA is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to \mathcal{P}(Q)$
- $\mathcal{P}(Q)$ is the power set of Q

NFAs accept an input string if *any* computation path ends in an accepting state. While more expressive and easier to design, NFAs have higher computational complexity ($O(2^n)$ in worst case).

6.3.1 Example 1: NFA for Strings Ending with '01'

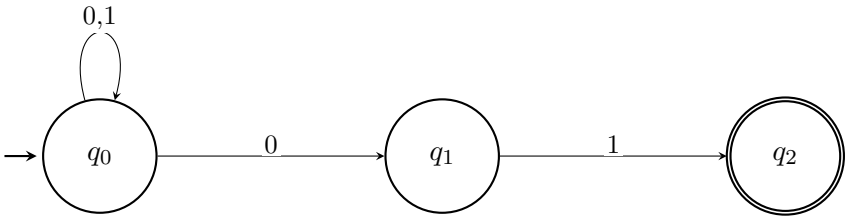


Figure 6.3: NFA accepting binary strings ending with '01'.

Execution Paths for "001":

- Path 1: $q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_0$ (reject)
- Path 2: $q_0 \xrightarrow{0} q_1 \xrightarrow{0}$ dead (reject)
- Path 3: $q_0 \xrightarrow{0} q_0 \xrightarrow{0} q_1 \xrightarrow{1} q_2$ (accept)

Key Features:

- ϵ-transitions allow state changes without input
- After 'a', machine nondeterministically chooses b-path or c-path
- Looping via ϵ-transitions enables repetition

6.4 NFA to DFA Conversion

The subset construction algorithm converts NFAs to equivalent DFAs:

1. DFA states = subsets of NFA states
2. Start state = ϵ-closure(q_0)
3. For each state S and symbol a:

$$T = \bigcup_{q \in S} \delta(q, a)$$

$$\delta_{DFA}(S, a) = \epsilon\text{-closure}(T)$$

4. Accept if subset contains any NFA accept state

6.4.1 Conversion Example 1: Simple NFA

Original NFA:

- States: {A, B, C}
- Transitions:
  - A $\xrightarrow{0}$ B
  - A $\xrightarrow{1}$ A

  - B $\xrightarrow{0}$ C
  - B $\xrightarrow{1}$ A
  - C $\xrightarrow{0}$ C
  - C $\xrightarrow{1}$ C
- Initial: A
- Accepting: C

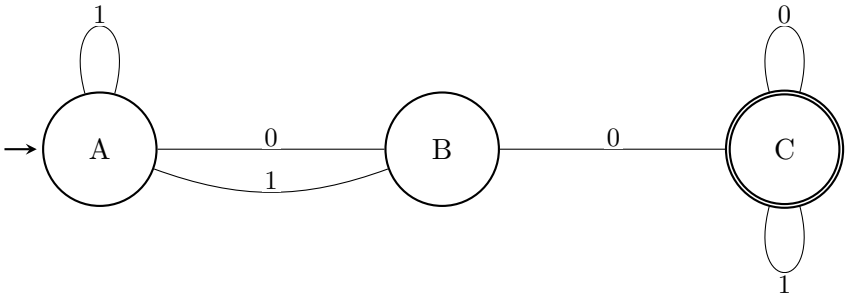


Figure 6.4: Original NFA for conversion example 1

Conversion Steps:

1. Start state: $\{A\}$
2. $\delta(\{A\}, 0) = \{B\}$
3. $\delta(\{A\}, 1) = \{A\}$
4. $\delta(\{B\}, 0) = \{C\}$

5. $\delta(\{B\}, 1) = \{A\}$
6. $\delta(\{C\}, 0) = \{C\}$
7. $\delta(\{C\}, 1) = \{C\}$

Resulting DFA:

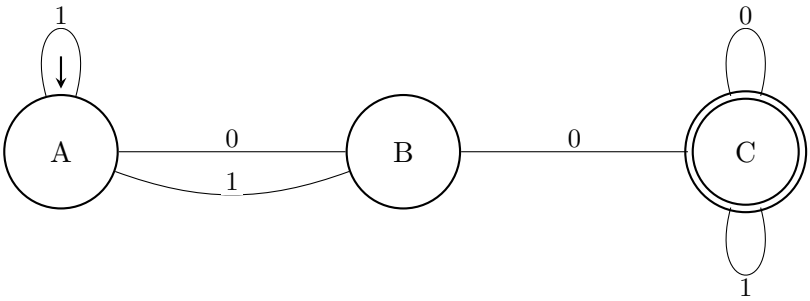


Figure 6.5: Equivalent DFA for the NFA

6.4.2 Programming Problems

1. **Syntax Table Generator**
   Write a program that categorizes characters in source code into:
   - Letters (a-z, A-Z)
   - Digits (0-9)
   - Symbols (, , ; etc.)

- Arithmetic operators (+, -, *, /)
- Logical operators (&&, ||, !)

Example:

```
Input: "int x = 10 + y;"
Output:
'i' : Letter
'n' : Letter
't' : Letter
' ' : Whitespace (ignore)
'x' : Letter
' ' : Whitespace
'=' : Symbol
' ' : Whitespace
'1' : Digit
'0' : Digit
' ' : Whitespace
'+' : Arithmetic Operator
' ' : Whitespace
'y' : Letter
';' : Symbol
```

2. **Tokenizer without strtok()**

Implement a tokenizer that:

- Reads source code
- Splits into tokens ignoring whitespace
- **Without using strtok() or similar library functions**
- Uses finite automata concepts

Example:

```
Input: "int x = -10 + y;"
Output:
int
x
=
-10
+
y
;
```

Sample Implementation:

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

```

```
4     void tokenize(const char *str) {
5         int state = 0;  // 0: whitespace, 1: token
6         for (int i = 0; str[i] != '\0'; i++) {
7             if (isspace(str[i])) {
8                 if (state == 1) {
9                     printf("\n");
10                    state = 0;
11                }
12            } else {
13                printf("%c", str[i]);
14                state = 1;
15            }
16        }
17        if (state == 1) printf("\n");
18    }
19
20    int main() {
21        char code[100] = "int x = -10 + y;";
22        tokenize(code);
23        return 0;
24    }
25
```

Automata Approach:

- States: WHITESPACE, TOKEN

- Transitions:
  - WHITESPACE + non-space → TOKEN (print char)
  - TOKEN + non-space → TOKEN (print char)
  - TOKEN + space → WHITESPACE (print newline)

Answer Key for Conceptual Questions

1. **Finite Automaton:** Mathematical model $(Q, \Sigma, \delta, q_0, F)$ with finite states, used in lexical analysis to recognize token patterns.

2. **DFA vs NFA:** DFA has exactly one transition per state-input pair, while NFA can have multiple. NFA allows ϵ-transitions and has higher theoretical complexity but easier pattern specification.

3. **NFA to DFA Conversion 1:**
   DFA States: {A}, {A,B}, {A,C}
   Transitions:
   - {A} + 0 → {A}, {A} + 1 → {A,B}
   - {A,B} + 0 → {A,C}, {A,B} + 1 → {A}

- {A,C} + 0 → {A}, {A,C} + 1 → {A,B,C} (accepting)
- {A,B,C} + 0 → {A,C}, {A,B,C} + 1 → {A,B,C}

4. **NFA to DFA Conversion 2:**
   DFA States: {A}, {A,B}, {A,C} (accepting)
   Transitions:
   - {A} + a → {A}, {A} + b → {A,B}
   - {A,B} + a → {A,C}, {A,B} + b → {A}
   - {A,C} + a → {A,C}, {A,C} + b → {A}

Conclusion

Finite automata provide the theoretical foundation for lexical analysis in compilers. While NFAs offer greater expressiveness and easier pattern specification, DFAs deliver the efficiency required for production compilers. The equivalence between these models (via subset construction) allows compiler designers to leverage both: using NFAs for initial specification and converting to DFAs for efficient execution.

Part III

Parsing Techniques

Chapter 7

Left Recursion and Factoring

7.1 Why is Left Recursion a Problem?

Left recursion occurs when a non-terminal calls itself on the left side of its production. This causes infinite recursion in top-down parsers like recursive descent parsers, making the parser enter an infinite loop.

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

For top-down parsing to work (such as in LL(1) parsing), grammars must not contain left recursion.
Therefore, we need to eliminate both direct and indirect left recursion to make grammars suitable for LL parsing.

7.1.1 Types of Left Recursion

There are two main types of left recursion that occur in context-free grammars:

Direct Left Recursion

Occurs when a non-terminal immediately refers to itself as the leftmost symbol in a production.

General Form:

$$A \to A\alpha_1 \mid A\alpha_2 \mid \cdots \mid A\alpha_n \mid \beta_1 \mid \beta_2 \mid \cdots \mid \beta_m$$

Where:

- α_i are non-empty strings of terminals and/or non-terminals
- β_j are strings that do not start with A

Indirect Left Recursion

Occurs through a chain of productions, where a non-terminal eventually derives itself as the leftmost symbol.

Example:

$$A \to B\alpha$$
$$B \to C\beta$$
$$C \to A\gamma \mid \delta$$

This creates a cycle $A \Rightarrow B\alpha \Rightarrow C\beta\alpha \Rightarrow A\gamma\beta\alpha$

Key Differences:

- Direct recursion is immediately visible in a single production
- Indirect recursion requires analyzing multiple productions to detect
- Both types are equally problematic for top-down parsers
- Elimination methods differ for each type (as shown in previous sections)

7.2 Direct Left Recursion

Example 1: Given a left-recursive production rule:

$$A \to A\alpha \mid \beta$$

Where:

- A is a non-terminal symbol
- α and β are strings of terminals and/or non-terminals
- β does not start with A

The equivalent right-recursive form after elimination is:

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

Example 2: Original Grammar with Left Recursion

$$A \to A\,p\,k \mid A\,b\,a \mid n\,y \mid z\,p \mid B$$

After Eliminating Left Recursion

$$A \to n\, y\, A' \mid z\, p\, A' \mid B\, A'$$
$$A' \to p\, k\, A' \mid b\, a\, A' \mid \varepsilon$$

Where:

- A' is a new non-terminal introduced in the transformation
- ε represents the empty string (epsilon production)
- The grammar now uses right recursion instead of left recursion

Example 3: Grammar with Left Recursion

$$E \to E + T \mid E - T \mid T$$
$$T \to T * F \mid T / F \mid F$$
$$F \to \text{id} \mid \text{num} \mid (\, E\,)$$

After Eliminating Left Recursion

$$
\begin{aligned}
E &\to T\, E' \\
E' &\to +\, T\, E' \mid \text{-}\, T\, E' \mid \varepsilon \\
T &\to F\, T' \\
T' &\to *\, F\, T' \mid /\, F\, T' \mid \varepsilon \\
F &\to \text{id} \mid \text{num} \mid (\, E\,)
\end{aligned}
$$

Where:

- E' and T' are new non-terminals introduced in the transformation
- ε represents the empty string (epsilon production)
- The grammar now uses right recursion instead of left recursion
- All original language constructs are preserved
- Operator precedence is maintained (*/ higher than +-)

7.3 Indirect Left Recursion

Example 1: Original Grammar with Indirect Recursion

$$S \to A\,b$$
$$A \to S\,a \mid c$$

Problem This grammar has indirect left recursion because:

- S can derive $A\,b$
- A can derive $S\,a$
- So $S \Rightarrow A\,b \Rightarrow S\,a\,b$ (left recursion!)

Solution Steps

1. Substitute S in A's production:

$$A \to (A\,b)\,a \mid c$$
$$\to A\,b\,a \mid c$$

2. Eliminate the new direct recursion in A:

$$A \to c\,A'$$
$$A' \to b\,a\,A' \mid \varepsilon$$

Final Transformed Grammar

$$S \to A\,b$$
$$A \to c\,A'$$
$$A' \to b\,a\,A' \mid \varepsilon$$

No infinite left recursion occurs!

Example 2: Original Grammar with Indirect Recursion

$$X \to Y\,m \mid n$$
$$Y \to X\,k \mid Z\,p$$
$$Z \to Y\,q \mid r$$

Transformation Steps

1. Substitute Y in Z's production:

$$Z \to (X\,k \mid Z\,p)\,q \mid r$$
$$\to X\,k\,q \mid Z\,p\,q \mid r$$

2. Eliminate direct recursion in Z:

$$\begin{aligned} Z &\to X\,k\,q\,Z' \mid r\,Z' \\ Z' &\to p\,q\,Z' \mid \varepsilon \end{aligned}$$

3. Substitute X in Y's production:

$$\begin{aligned} Y &\to (Y\,m \mid n)\,k \mid Z\,p \\ &\to Y\,m\,k \mid n\,k \mid Z\,p \end{aligned}$$

4. Eliminate direct recursion in Y:

$$\begin{aligned} Y &\to n\,k\,Y' \mid Z\,p\,Y' \\ Y' &\to m\,k\,Y' \mid \varepsilon \end{aligned}$$

5. Final grammar:

$$\begin{aligned} X &\to Y\,m \mid n \\ Y &\to n\,k\,Y' \mid Z\,p\,Y' \\ Y' &\to m\,k\,Y' \mid \varepsilon \\ Z &\to X\,k\,q\,Z' \mid r\,Z' \\ Z' &\to p\,q\,Z' \mid \varepsilon \end{aligned}$$

Example 3: Original Grammar with Indirect Recursion

$$S \to A\,f \mid b$$
$$A \to A\,c \mid S\,d \mid B\,c$$
$$B \to A\,g \mid S\,h \mid k$$

Transformation Steps

$$A \to A\,c \mid A\,f\,d \mid b\,d \mid B\,c$$
$$A \to b\,d\,A' \mid B\,c\,A'$$
$$A' \to c\,A' \mid f\,d\,A' \mid \varepsilon$$

$$B \to A\,g \mid S\,h \mid k$$
$$B \to b\,d\,A'\,g \mid B\,c\,A'\,g \mid A\,f\,h \mid b\,h \mid k$$
$$B \to b\,d\,A'\,g \mid B\,c\,A'\,g \mid b\,d\,A\,f\,h \mid B\,c\,A'\,f\,h \mid b\,h \mid k$$
$$B \to b\,d\,A'\,g\,B' \mid b\,d\,A\,f\,h\,B' \mid b\,h\,B' \mid k\,B'$$
$$B' \to c\,A'\,g\,B' \mid c\,A'\,f\,h\,B' \mid \varepsilon$$

Where:

- A' and B' are new non-terminals introduced to eliminate recursion

- ε represents the empty string
- The grammar maintains the same language while removing indirect left recursion

Final Transformed Grammar

$$
\begin{aligned}
S &\to A\,f \mid b \\
A &\to b\,d\,A' \mid B\,c\,A' \\
A' &\to c\,A' \mid f\,d\,A' \mid \varepsilon \\
B &\to b\,d\,A'\,g\,B' \mid b\,d\,A\,f\,h\,B' \mid b\,h\,B' \mid k\,B' \\
B' &\to c\,A'\,g\,B' \mid c\,A'\,f\,h\,B' \mid \varepsilon
\end{aligned}
$$

7.4 What is Left Factoring?

Left factoring is a grammar transformation technique that removes ambiguity by factoring out common prefixes from production rules. It's particularly useful for:

- Making grammars suitable for top-down parsing
- Eliminating FIRST-FIRST conflicts in LL(k) parsers
- Resolving predictive parsing ambiguities

7.4.1 When to Apply

Apply when a non-terminal has two or more productions with common prefixes:

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

where α is the common prefix.

7.4.2 Algorithm

1. Identify productions with common prefixes
2. Extract the common prefix α
3. Create new productions for the remaining parts

General form:

$$A \to \alpha, \beta_1 \mid \alpha, \beta_2 \mid \gamma_1 \mid \gamma_2 \mid \alpha\beta + 3$$
$$A \to \alpha A' \mid \gamma_1 \mid \gamma_2$$
$$A' \to \beta_1 \mid \beta_2 \mid \beta_3$$

Example 1: Basic Left Factoring

Original Grammar

$$S \to aB \mid aC$$
$$B \to b$$
$$C \to c$$

Problem

Both S productions start with 'a' - cannot predict which to choose.

Transformed Grammar

$$
\begin{aligned}
S &\to aS' \\
S' &\to B \mid C \\
B &\to b \\
C &\to c
\end{aligned}
$$

Example 2: Complex Case

Original Grammar

$$
\begin{aligned}
Expr &\to id + Expr \\
&\to id - Expr \\
&\to id
\end{aligned}
$$

Transformed Grammar

$$
\begin{aligned}
Expr &\to id\ Expr' \\
Expr' &\to +Expr \\
&\to -Expr \\
&\to \varepsilon
\end{aligned}
$$

Example 3:

Original Grammar

$$E \rightarrow edu \mid edu.bd \mid diu.edu \mid diu.bd \mid diu \mid education$$
$$\mid educate \mid educated \mid twinkle \mid humty \mid dumty$$

Transformed Grammar

$$E \rightarrow eduE^{i} \mid diu.edu \mid diu.bd \mid diu \mid twinkle \mid humty \mid dumty$$
$$E^{i} \rightarrow \varepsilon \mid .bd \mid cation \mid cate \mid cated$$
$$E \rightarrow eduE^{i} \mid diuE^{ii} \mid twinkle \mid humty \mid dumty$$
$$E^{ii} \rightarrow \varepsilon \mid .edu \mid .bd$$
$$E^{i} \rightarrow cateE^{iii} \mid .bd \mid \varepsilon$$
$$E^{iii} \rightarrow ion \mid e \mid ed$$
$$E^{iii} \rightarrow eE^{iv} \mid ion$$
$$E^{iv} \rightarrow \varepsilon \mid d$$
$$E^{ii} \rightarrow \varepsilon \mid .E^{v}$$
$$E^{v} \rightarrow edu \mid bd$$

Chapter 8

First() and Follow() Sets for LL(1) Parsing

8.1 Introduction

In LL(1) parsers, the **FIRST()** and **FOLLOW()** sets are used to construct the parsing table. These sets help determine which production to apply based on the current input symbol and non-terminal at the top of the parsing stack.

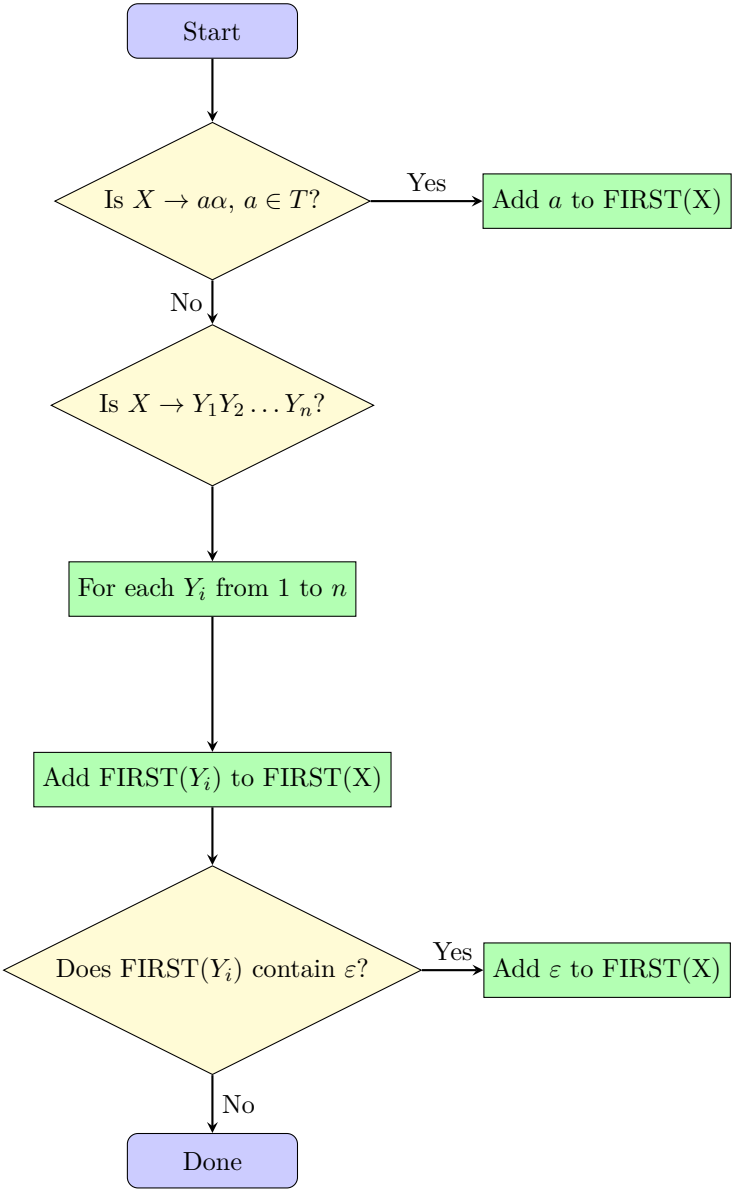
8.2 FIRST() Set

Definition: The FIRST set of a symbol α (terminal, non-terminal, or string) is the set of terminals that begin the strings derivable from α.

Rules for Computing FIRST Sets

- If X is a terminal, then FIRST(X) = $\{X\}$.
- If X is a non-terminal and $X \to \varepsilon$ is a production, then add ε to FIRST(X).
- If $X \to Y_1 Y_2 \ldots Y_k$:
  - Add all non-ε symbols of FIRST(Y_1) to FIRST(X)
  - If $\varepsilon \in$ FIRST(Y_1), continue to check FIRST(Y_2), and so on
  - If ε is in all FIRST(Y_i), then add ε to FIRST(X)

FIRST(X) Flowchart



8.3 FOLLOW() Set

Definition: The FOLLOW set of a non-terminal A is the set of terminals that can appear immediately to the right of A in some sentential form.

Rules for Computing FOLLOW Sets

- Add \$ (end of input) to FOLLOW(S) where S is the start symbol.
- For any production $A \to \alpha B \beta$:
  - Add all non-ε symbols of FIRST(β) to FOLLOW(B)
  - If $\varepsilon \in$ FIRST(β), then add FOLLOW(A) to FOLLOW(B)
- For production $A \to \alpha B$, add FOLLOW(A) to FOLLOW(B)

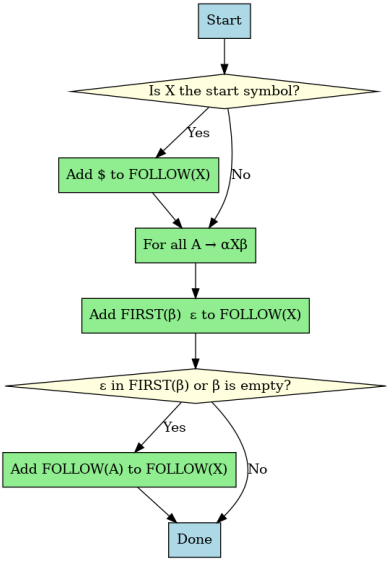


Figure 8.1: Construction Flowchart of FOLLOW()

FOLLOW(X) Set Construction Flowchart

8.4 Example:

Example 01:

$$
\begin{aligned}
Grammar : E &\rightarrow T\,E' \\
E' &\rightarrow +\,T\,E' \mid \varepsilon \\
T &\rightarrow F\,T' \\
T' &\rightarrow *\,F\,T' \mid \varepsilon \\
F &\rightarrow (\,E\,) \mid id
\end{aligned}
$$

Step 1: FIRST Sets

- FIRST(**id**) = { id }
- FIRST(**(**) = { (}
- FIRST(F) = FIRST(**(E)**) ∪ FIRST(**id**) = { (, id }
- FIRST(T) = FIRST(F T') = FIRST(F) = { (, id }
- FIRST(E) = FIRST(T E') = FIRST(T) = { (, id }
- FIRST(E') = FIRST(+ T E') ∪ FIRST(ε) = { +, ε}
- FIRST(T') = FIRST(* F T') ∪ FIRST(ε) = { *, ε }

FIRST Sets Summary

$$\begin{aligned}
\text{FIRST}(E) &= \{(, id\} \\
\text{FIRST}(E') &= \{+, \varepsilon\} \\
\text{FIRST}(T) &= \{(, id\} \\
\text{FIRST}(T') &= \{*, \varepsilon\} \\
\text{FIRST}(F) &= \{(, id\}
\end{aligned}$$

Step 2: FOLLOW Sets

- Start symbol: FOLLOW(E) = { $ }

- From $E \rightarrow T\ E'$, FOLLOW(T) $\supseteq$ FIRST(E') - {ε} = { + }
- From $E \rightarrow T\ E'$, FOLLOW(E') $\supseteq$ FOLLOW(E) = { $ }
- From $E' \rightarrow +\ T\ E'$, FOLLOW(T) $\supseteq$ FIRST(E') - {ε} = { + }, but we already have that from earlier.
- From $E' \rightarrow +\ T\ E'$, FOLLOW(E') $\supseteq$ FOLLOW(E') = { $ }
- From $T \rightarrow F\ T'$, FOLLOW(F) $\supseteq$ FIRST(T') - {ε} = { * }
- From $T \rightarrow F\ T'$, FOLLOW(T') $\supseteq$ FOLLOW(T) = { +, $ }
- From $T' \rightarrow *\ F\ T'$, FOLLOW(F) $\supseteq$ FIRST(T') - {ε} = { * }, already included
- From $T' \rightarrow *\ F\ T'$, FOLLOW(T') $\supseteq$ FOLLOW(T') = { +, $ }
- From $F \rightarrow (E)$, FOLLOW(E) $\supseteq$ {) }

FOLLOW Sets Summary

$$\text{FOLLOW}(E) = \{), \$\}$$
$$\text{FOLLOW}(E') = \{), \$\}$$
$$\text{FOLLOW}(T) = \{+,), \$\}$$
$$\text{FOLLOW}(T') = \{+,), \$\}$$
$$\text{FOLLOW}(F) = \{*, +,), \$\}$$

Non-terminal	FIRST	FOLLOW
E	{ (, id }	{), $ }
E'	{ +, ε }	{), $ }
T	{ (, id }	{ +,), $ }
T'	{ *, ε }	{ +,), $ }
F	{ (, id }	{ *, +,), $ }

8.5 LL(1) Parsing Table Construction

To fill the LL(1) parsing table:

- For each production $A \to \alpha$:
  - For each terminal a in FIRST(α), add $A \to \alpha$ to table[A, a]
  - If $\varepsilon \in$ FIRST(α), then for each terminal b in FOLLOW(A),

add $A \to \varepsilon$ to table[A, b]

LL(1) Parsing Table

Non-terminal	**(**	**id**	**+**	*****	**)**	**$**
E	$E \to TE'$	$E \to TE'$				
E'			$E' \to +TE'$		$E' \to \varepsilon$	$E' \to \varepsilon$
T	$T \to FT'$	$T \to FT'$				
T'			$T' \to \varepsilon$	$T' \to *FT'$	$T' \to \varepsilon$	$T' \to \varepsilon$
F	$F \to (E)$	$F \to id$				

Summary

- FIRST sets help determine what a non-terminal can derive.
- FOLLOW sets help determine what can follow a non-terminal.
- FIRST and FOLLOW sets are crucial for LL(1) parser construction.

Example 02:

$$S \to Bb \mid cd$$
$$B \to aB \mid \varepsilon$$
$$C \to cC \mid \varepsilon$$

FIRST and FOLLOW Sets

$$\text{FIRST}(S) = \{a, b, c\}$$
$$\text{FIRST}(B) = \{a, \varepsilon\}$$
$$\text{FIRST}(C) = \{c, \varepsilon\}$$

$$\text{FOLLOW}(S) = \{\$\}$$
$$\text{FOLLOW}(B) = \{b\}$$
$$\text{FOLLOW}(C) = \{\} \quad \text{(not used)}$$

FIRST and FOLLOW Sets

Non-terminal	FIRST	FOLLOW
S	{a, b, c}	{$}
B	{a, ε}	{b}
C	{c, ε}	{ } (not used)

LL(1) Parsing Table

Non-terminal	a	b	c	d	$
S	$S \to Bb$		$S \to cd$		
B	$B \to aB$	$B \to \varepsilon$			
C			$C \to cC$		

Example 03:

$$S \rightarrow ABCDE$$

$$A \rightarrow a \mid \varepsilon$$

$$B \rightarrow b \mid \varepsilon$$

$$C \rightarrow c$$

$$D \rightarrow d \mid \varepsilon$$

$$E \rightarrow e \mid \varepsilon$$

FIRST and FOLLOW Sets

$$\text{FIRST}(S) = \{a, b, c\}$$
$$\text{FIRST}(A) = \{a, \varepsilon\}$$
$$\text{FIRST}(B) = \{b, \varepsilon\}$$
$$\text{FIRST}(C) = \{c\}$$
$$\text{FIRST}(D) = \{d, \varepsilon\}$$
$$\text{FIRST}(E) = \{e, \varepsilon\}$$

$$\text{FOLLOW}(S) = \{\$\}$$
$$\text{FOLLOW}(A) = \{b, c\}$$
$$\text{FOLLOW}(B) = \{c\}$$
$$\text{FOLLOW}(C) = \{d, e, \$\}$$
$$\text{FOLLOW}(D) = \{e, \$\}$$
$$\text{FOLLOW}(E) = \{\$\}$$

FIRST and FOLLOW Sets

Non-terminal	FIRST	FOLLOW
S	{a, b, c}	{$}
A	{a, ε}	{b, c}
B	{b, ε}	{c}
C	{c}	{d, e, $}
D	{d, ε}	{e, $}
E	{e, ε}	{$}

LL(1) Parsing Table

Non-terminal	a	b	c	d	e	$
S	$S \to ABCDE$	$S \to ABCDE$	$S \to ABCDE$			
A	$A \to a$	$A \to \varepsilon$	$A \to \varepsilon$			
B		$B \to b$	$B \to \varepsilon$			
C			$C \to c$			
D				$D \to d$	$D \to \varepsilon$	$D \to \varepsilon$
E					$E \to e$	$E \to \varepsilon$

Chapter 9

LR(0) Parser

9.1 Introduction to Bottom-Up Parsing

Bottom-up parsing, also known as shift-reduce parsing, constructs the parse tree from leaves to root. LR(0) parsing is a fundamental bottom-up technique that:

- Processes input from **Left** to right
- Constructs a **Rightmost** derivation in reverse
- Uses **0** lookahead symbols

LR(0) parsers are particularly valuable because they:

- Can recognize a larger class of grammars than LL parsers
- Provide systematic error detection
- Serve as the foundation for more advanced parsers (SLR, LALR)
- Are used in production compilers like GCC and Clang

The LR parsing algorithm was first introduced by Donald Knuth in 1965 and remains a cornerstone of modern compiler design due to its efficiency and predictive power.

9.2 LR Parser Fundamentals

The LR parser is an efficient bottom-up syntax analysis technique that can be used for a large class of context-free grammars. This technique is also called **LR(0)** parsing, where:

- **L** stands for left-to-right scanning of input
- **R** stands for rightmost derivation in reverse
- **0** stands for number of input symbols of lookahead

9.3 Augmented Grammar

The construction of an LR parser begins with creating an augmented grammar:

- If G is a grammar with starting symbol S
- Then G' (augmented grammar for G) has:
  - New starting symbol S'
  - Production rule: $S' \to \cdot S$

The purpose of augmentation is to:

- Provide a clear starting point for parsing
- Indicate when parsing should terminate
- The dot ($\cdot$) notation represents the parsing progress:
  - Left of dot: already processed by compiler
  - Right of dot: remaining input to process

9.4 Constructing the LR Parsing Table

The systematic construction of an LR parsing table involves:

Step 1: **Augment the grammar**

- Add $S' \to \cdot S$ production
- Mark this as the initial state

Step 2: **Build LR(0) collection of items**

- Compute closure of kernel items
- Determine state transitions (goto operations)
- Identify reduce/reduce and shift/reduce conflicts

Step 3: **Define parsing table functions**

- `action`(list of terminals): Determines shift/reduce actions
- `goto`(list of non-terminals): Specifies state transitions

Given Grammar

$$S \to AA$$
$$A \to aA \mid b$$

Solution

Step 1: Augmented Grammar

The augmented grammar is:

$$S' \to .S \quad \text{[0th production]}$$
$$S \to .AA \quad \text{[1st production]}$$
$$A \to .aA \quad \text{[2nd production]}$$
$$A \to .b \quad \text{[3rd production]}$$

Step 2: LR(0) Collection of Items

The transitions are:

- $\mathrm{I}_0 \to \mathrm{I}_1$ when $S' \to S.$ (accept state)
- $\mathrm{I}_0 \to \mathrm{I}_2$ when $S \to A.A$
- $\mathrm{I}_0 \to \mathrm{I}_3$ when $A \to a.A$
- $\mathrm{I}_0 \to \mathrm{I}_4$ when $A \to b.$
- $\mathrm{I}_2 \to \mathrm{I}_5$ when $S \to AA.$
- $\mathrm{I}_2 \to \mathrm{I}_3$ when $A \to a.A$
- $\mathrm{I}_2 \to \mathrm{I}_4$ when $A \to b.$

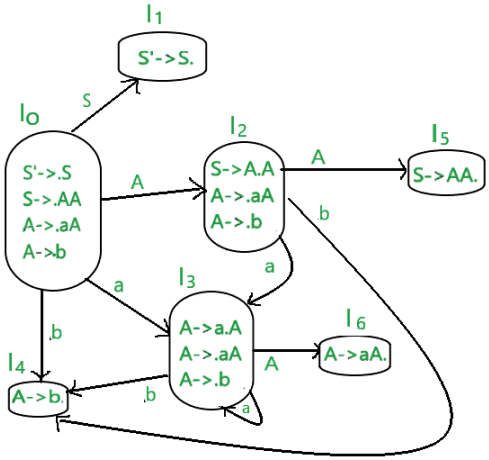


Figure 9.1: LR(0) items automaton (diagram would be included here)

- $I_3 \rightarrow I_4$ when $A \rightarrow b.$
- $I_3 \rightarrow I_6$ when $A \rightarrow aA.$
- $I_3 \rightarrow I_3$ when $A \rightarrow a.A$

Step 3: Parsing Table Construction

Where:

- sn means shift to state n

Table 9.1: LR(0) Parsing Table

State	Action		Goto		
	a	b	$	S	A
0	s3	s4		1	2
1			acc		
2	s3	s4			5
3	s3	s4			6
4	r3	r3	r3		
5	r1	r1	r1		
6	r2	r2	r2		

- rn means reduce by production n
- acc means accept
- Blank entries indicate error

Part IV

Code Generation and Optimization

Chapter 10

Intermediate Code Generation

Introduction

Intermediate code generation translates source code into a machine-independent representation (IR). This IR serves as a bridge between high-level languages and machine code. Key benefits include:

- **Machine Independence**: IR can be targeted to multiple architectures

- **Optimization Enablement**: Facilitates machine-independent code improvements
- **Portability**: Allows compiler reuse across different systems
- **Structural Simplification**: Separates front-end and back-end concerns
- **Efficient Translation**: Provides clear structure for code generation

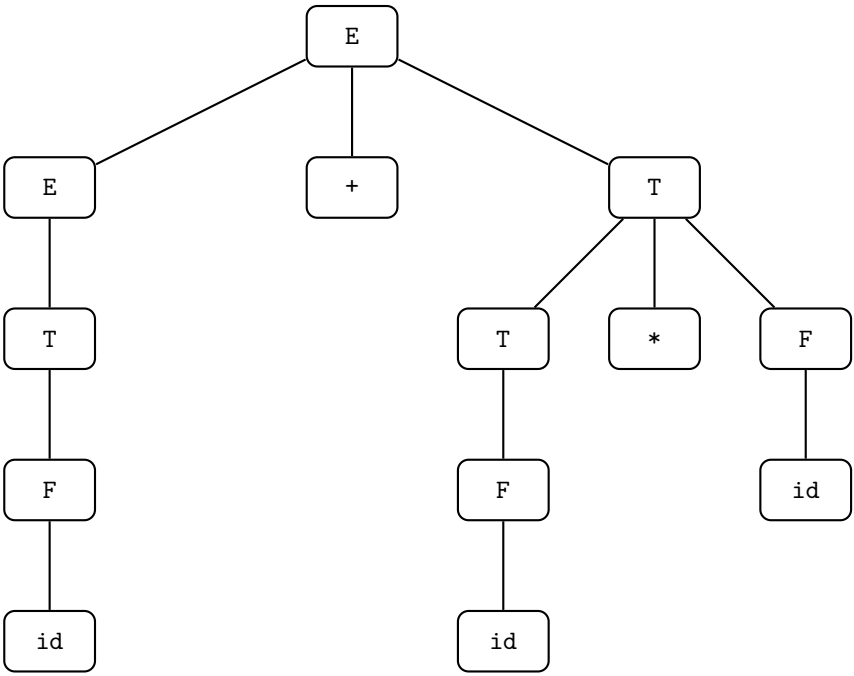
Common IR forms include three-address code, syntax trees, and Directed Acyclic Graphs (DAGs).

10.1 Syntax Trees vs Parse Trees

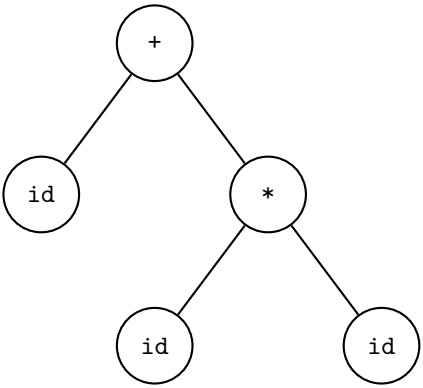
Conceptual Differences

- **Parse Trees**: Concrete representation showing complete derivation using grammar rules
- **Syntax Trees**: Abstract representation capturing program structure without redundant nodes

Parse Tree for String: `id + id * id`
Grammar: $E \to E{+}T \mid T$, $T \to T{*}F \mid F$, $F \to (E) \mid \mathbf{id}$


E
E
+
T
T
T
*
F
F
F
id
id
id

Syntax Tree for `id + id * id`



Comparative Analysis

Aspect	Parse Tree	Syntax Tree
Representation	Concrete syntax derivation	Abstract program structure
Size	Larger (includes all grammar symbols)	Compact (essential nodes only)
Usage	Syntax validation	Semantic analysis and code generation
Redundancy	High (shows derivation steps)	Minimal (focuses on operations)
Construction	Built during parsing	Derived from parse tree

10.2 Directed Acyclic Graphs (DAGs)

Concept and Purpose

A DAG is a graphical representation that:

- Identifies and shares common subexpressions
- Eliminates redundant computations
- Optimizes expression evaluation
- Preserves operation order through directed edges

Construction Rules

1. Create leaf nodes for variables and constants
2. For each operation:
   - Check for existing identical node
   - Create new node only if necessary
3. Maintain topological order (no cycles)
4. Propagate constant values
5. Annotate nodes with computed values

DAG Examples Without Figures

Example 1: Expression: $a + a + a$

DAG Structure:

- Single leaf node for variable a
- First addition: $t1 = a + a$ (new node)
- Second addition: $t2 = t1 + a$ (new node)
- No common subexpressions beyond a

Optimization Insight:
While the variable a is reused, the additions cannot be combined as they are sequential operations.

Example 2: Expression: $(((a + a) + (a + a)) + ((a + a) + (a + a)))$

DAG Structure:

- Single leaf node for variable a
- Single addition node for $a + a$ (reused 8 times)
- Higher-level additions reuse intermediate results
- Only 3 unique nodes: leaf a, addition node, and root node

Optimization Insight:
Extreme redundancy elimination - 16 occurrences of a reduced to 1 leaf, 8 additions reduced to 1 computation.

Example 3: Expression: $(a + b) * (a + b + c)$

DAG Structure:

- Leaf nodes for a, b, c
- Single addition node for $a + b$ (reused)
- Addition node for $(a + b) + c$ using previous result
- Multiplication node for $(a + b) * ((a + b) + c)$

Optimization Insight:
The common subexpression $(a + b)$ is computed once and reused, reducing the total operations.

10.3 Three-Address Code (TAC)

Characteristics

- Maximum of three operands per instruction
- Simple, linear representation
- Easy to generate and optimize
- Machine-independent format

Instruction Types

Type	Format	Example
Binary Operation	`x = y op z`	`t1 = b * c`
Unary Operation	`x = op y`	`t2 = -t1`
Copy	`x = y`	`a = t2`
Conditional Jump	`if x relop y goto L`	`if a < b goto L1`
Unconditional Jump	`goto L`	`goto L2`
Label	`L:`	`L1:`

Example 1: `a = b + c * d`

```
t1 = c * d      // Multiply c and d
t2 = b + t1     // Add b and t1
a = t2          // Assign to a
```

Example 2: `if (a < b) x = y + z`

```
  if a < b goto L1  // Conditional jump
  goto L2            // Else branch
L1:                 // True case label
  t1 = y + z        // Compute y+z
  x = t1            // Assign to x
L2:                 // Exit label
```

10.4 TAC Implementation Methods

Quadruples Representation

Index	Op	Arg1	Arg2	Result
1	*	c	d	t1
2	+	b	t1	t2
3	=	t2		a
4	<	a	b	L1
5	jmp			L2
6	+	y	z	t3
7	=	t3		x

Advantages: Explicit results, easy rearrangement

Triples Representation

Index	Op	Arg1	Arg2
1	*	c	d
2	+	b	(1)
3	=	a	(2)
4	<	a	b
5	jmp		
6	+	y	z
7	=	x	(6)

Advantages: Compact, uses pointers to results

Indirect Triples

Pointer	Instruction
100	(1) * c d
101	(2) + b (1)
102	(3) = a (2)
103	(4) < a b
104	(5) jmp L2
105	(6) + y z
106	(7) = x (6)

Advantages: Easy code motion during optimization

10.5 Operator Precedence and Associativity

Precedence Hierarchy

Precedence	Operators	Description
1	`( )`	Parentheses (highest)
2	`* / %`	Multiplicative operators
3	`+ -`	Additive operators
4	`< <= > >=`	Relational operators
5	`== !=`	Equality operators
6	`&&`	Logical AND
7	`||`	Logical OR
8	`= += -= *= /=`	Assignment (lowest)

Associativity Rules

Operator Type	Associativity
Arithmetic (`+ - * /`)	Left to Right
Relational (`< <= > >=`)	Left to Right
Equality (`== !=`)	Left to Right
Logical (`&& ||`)	Left to Right
Assignment (`= += -=`)	Right to Left
Unary (`! - ++ -`)	Right to Left

Key Applications:

- Determines evaluation order in expressions
- Resolves ambiguity in expression parsing
- Guides intermediate code generation
- Ensures correct computation semantics

10.6 Programming Problems

P 10.1: Extract Prepositions from a String

Write a C program that extracts prepositions from a string. It:

- Uses `strtok()` to split input into words
- Converts words to lowercase for case-insensitive matching
- Checks against a predefined preposition list
- Outputs identified prepositions

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

const char *prepositions[] = {
    "in", "on", "at", "by", "with", "about", "against",
    "between", "into", "through", "during", "before",
    "after", "above", "below", "to", "from", "up", "down",
    "under", "over", "among", "of", "for", "since",
    "without", "within", "upon"
};

const int prepCount = sizeof(prepositions) /
                      sizeof(prepositions[0]);

int isPreposition(char *word) {
```

```
    for (int i = 0; i < prepCount; i++) {
        if (strcmp(word, prepositions[i]) == 0)
            return 1;
    }
    return 0;
}

int main() {
    char str[200], word[50];
    printf("Enter sentence: ");
    fgets(str, sizeof(str), stdin);
    str[strcspn(str, "\n")] = '\0';

    printf("Prepositions: ");
    char *token = strtok(str, " ");
    while (token) {
        for (int i = 0; token[i]; i++)
            word[i] = tolower(token[i]);

        if (isPreposition(word))
            printf("%s ", word);

        token = strtok(NULL, " ");
    }
    printf("\n");
    return 0;
}
```

Sample Input/Output

Input	Output
Enter sentence: She sat on the chair with a book in her hand	Prepositions: on with in
Enter sentence: The cat jumped over the fence before dinner	Prepositions: over before
Enter sentence: He traveled through mountains during summer	Prepositions: through during

Key Features

- **Preposition List**: 28 common English prepositions
- **Case Handling**: Converts to lowercase for case-insensitivity
- **Tokenization**: Splits input using `strtok()`
- **Efficiency**: Linear search through preposition list
- **Safety**: Input buffer size limited to 200 characters

10.7 Exercises

Conceptual Questions

1. **Difference between parse tree and syntax tree**:
   - Parse tree shows complete derivation using grammar rules

- Syntax tree abstracts grammatical details, focuses on operations
- Parse trees include non-terminals, syntax trees only essential nodes
- Parse trees larger with redundant information

2. **Construct syntax trees for**:
   (a) `(a + b) * (c - d) + ((e / f) * (a + b))`
   (b) `(a + a) + b * c + (b * c + c) + (d + d + d + d)`
   (c) `b * -c + b * -c`
   (d) `a + a * (b - c) + (b - c) * d`
   (e) `(a + b) * (c + d) + (a + b + c)`
3. **Construct DAG for**:
   (a) `(a + b) × (a + b + c)`
   (b) `a + a + a + a`
   (c) `a + a + a + a + a`
4. **Generate intermediate code for**:
   (a) `a + b * c - d / (b * c)`
   (b) `(-c * b) + (-c * d)`
   (c) `(x + y) * (y + z) + (x + y + z)`
   (d) `(a × b) + (c + d) - (a + b + c + d)`
   (e) `if A < B and C < D then t = 1 else t = 0`

Programming Problems

1. **Preposition Count Program**: Write a C program that:

- Counts prepositions in input strings
- Uses `strtok()` for word tokenization
- Converts words to lowercase for case-insensitive matching
- Compares against predefined preposition list
- Outputs total count and list of found prepositions

Extension: Modify to handle punctuation and contractions

Chapter 11

Mini Interpreter for a Simple Programming Language

11.1 Project Overview

This project implements a basic interpreter for a custom programming language using C. The interpreter demonstrates fundamental language processing techniques including lexical analysis, parsing, and execution. Key components include:

- Variable storage using a symbol table
- Command parsing and execution
- Expression evaluation
- Conditional statement handling
- Interactive REPL (Read-Eval-Print Loop) interface

11.2 Core Components

11.2.1 Symbol Table

Variables are stored in a fixed-size array acting as a symbol table. Each entry contains:

- Variable name (string)
- Integer value
- Case-sensitive lookup

Linear search handles variable resolution with O(n) time complexity.

11.2.2 Command Processing

The interpreter supports these commands:

`let` Variable assignment with validation

`print` Outputs values or string literals

`if` Conditional execution with comparison operators

`exit` Terminates the interpreter

11.2.3 Parsing Strategy

1. Input trimming and tokenization
2. Command type detection
3. Expression evaluation
4. Recursive command execution for conditionals

11.3 Key Features

- Interactive command prompt
- Dynamic variable creation/updating
- Support for comparison operators: `>`, `<`, `==`, `!=`, `>=`, `<=`
- Error handling for common issues
- Recursive conditional execution

11.4 Implementation Challenges

- Handling whitespace in commands
- Differentiating between variables and literals
- Managing recursive command execution
- Validating variable names
- Implementing robust error messages

Recursion in conditional execution was particularly challenging to implement safely without stack overflow vulnerabilities.

11.5 Design Tradeoffs

Array-based storage: Simplicity for small-scale implementation

Linear search: Acceptable for limited variables

Recursive execution: Natural mapping to language semantics

Fixed-size buffers: Balance between safety and simplicity

11.6 Learning Outcomes

This project demonstrates:

- Fundamental interpreter architecture
- Symbol table management techniques
- Recursive parsing strategies
- Tradeoffs in language design
- Error handling in language processors

- Memory management considerations

11.7 Project Repository

QR Code

Access source code and documentation: https://github.com/MRA-Sami/Compiler_Design_Book

11.8 Extensions Future Work

Potential enhancements include:

- Arithmetic expression evaluation
- Loop constructs
- Function definitions
- Type system expansion
- Bytecode compilation
- Garbage collection

- Persistent variable storage

11.9 Conclusion

This interpreter demonstrates core language processing concepts while maintaining simplicity. The implementation provides a foundation for exploring more advanced language features and optimization techniques. The project highlights the relationship between language design decisions and implementation complexity.

References

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2nd Edition)*. Pearson, 2006.
2. Kenneth C. Louden. *Compiler Construction: Principles and Practice*. Cengage Learning, 1997.
3. GeeksforGeeks. https://www.geeksforgeeks.org/compiler-design-tutorial/
4. TutorialsPoint. https://www.tutorialspoint.com/compiler_design/index.htm
5. Programiz. https://www.programiz.com/compiler-design

Note to Readers

This is our very first attempt at creating a comprehensive book on **Compiler Design**. While we have made every effort to ensure accuracy and clarity, we acknowledge that mistakes or areas for improvement may still exist.

If you come across any errors, omissions, or have constructive feedback, we would greatly appreciate your input. Your suggestions will help us improve and release better editions in the future.

For any updates, corrections, or feedback, please contact us at the email address provided on the second page of this book.

Thank you for your understanding and support!

This book will develop you a hanside deep undergraduate their compiler tutor. It is a bejenner's guide for step-by-step tutors under

> “The real world doesn't reward perfectionists. It rewards people who get things done.”

GitHUB Repository

Book PDF