COMPILER DESIGN NOTE BOOK

A SIMPLE HANDBOOK ON COMPILER THEORY AND PRACTICAL APPROACHES

"The art of Compiler Design is the art of making language real for the machine."

BY SAJJADUL ISLAM SOMON

Compiler Design

NOTE BOOK

A Simple Handbook on Compiler Theory and Practical Approaches

Academic Session: Spring 2025
Course Title: Compiler Design & Lab
Course Code: CSE331 & 332

By
Sajjadul Islam Somon

Department of Computer Science and Engineering
Daffodil International University

“The art of Compiler Design is the art of making language real for the machine.”

Edition & Copyright

ISBN: N/A
Hardcover: N/A
Paperback: N/A
eBook: Checkout my GitHub

First Edition: December 12, 2024
Printed in Bangladesh

Publication & Distribution:
JOYA Trade International
35 City Corp., Nilkhet Main Road, Dhaka, Bangladesh
Website: www..com
Linkedin: *linkedin.com/in/sajjadul-islam-somon/*
Facebook: *facebook.com/sajjadul.islam.somon/*
Email: *somon15-5749@diu.edu.bd*

Copyright Notice

About This Book

This book serves as a comprehensive and practical guide to the laboratory component of the Compiler Design course. It combines foundational theory with hands-on coding exercises to help students understand how compilers are built from the ground up.

Each lab work focuses on a specific concept in compiler construction—such as lexical analysis, parsing, grammar transformations, or intermediate code generation—and is paired with relevant C programming implementations. To reinforce learning, every lab includes practice problems and viva questions.

By the end of this book, readers will have explored and implemented essential components of a compiler, gaining both theoretical insight and practical experience.

Author's Note

This book has been prepared as part of the Compiler Design Lab assignment, under the guidance of Associate Professor Mushfiqur Rahman, CSE, DIU.

As a student of Computer Science and Engineering, I approached this project with a focus on clarity, practicality, and academic rigor. The aim is to simplify complex compiler concepts through hands-on lab work, coding tasks, and structured learning. I hope this manual not only fulfills its academic purpose but also serves as a helpful companion for anyone exploring the fascinating world of compiler design

— Sajjadul Islam Somon

Contents Overview

CONTENTS

Compiler Design

THEORY PART

1

Compiler Design—Introduction

Definition

Compiler Design refers to the study and creation of software that translates high-level programming languages (like C, C++, Java) into low-level machine code or intermediate code that a computer can execute.

Background

In the early stages of computer science, programming was done using machine language, which consists of binary instructions (0s and 1s) that directly control the CPU. These were extremely low-level, error-prone, and difficult to understand, requiring a deep knowledge of hardware architecture.

To simplify this process, Assembly Language was introduced. Assembly provided a slightly more readable syntax using mnemonics like MOV, ADD, SUB, etc. Each assembly instruction maps directly to a single machine instruction.

Limitations of Assembly Language:

- Despite being a step forward, assembly had its drawbacks:
- It was still hardware-specific, lacking portability.
- Programming large-scale applications was cumbersome and error-prone.
- No concept of high-level abstractions like functions, loops, or conditionals.
- Required detailed management of memory and registers.

The Rise of High-Level Languages:

To overcome these limitations, high-level programming languages (HLLs) like FORTRAN, COBOL, C, Pascal, and later Java, Python, etc., were developed.
These languages introduced:

- Abstraction from hardware (portable code).
- Use of natural language-like syntax (e.g., if, while, for).
- Support for modularity, data structures, type systems.
- Increased productivity and readability.

However, high-level languages are not directly executable by the machine.

The Need for Translation

Because the CPU can only understand machine code, there arose a need to translate high-level programs into machine-executable code. This is where compilers, interpreters, and assemblers became essential.

Compiler: Translates high-level code to machine code.
Interpreter: Executes code line-by-line.
Assembler: Translates assembly to machine code.

Summary of the Evolution

Stage Language Type Characteristics

Stage	Language Type	Characteristics
Machine Code	Binary (0s and 1s)	Low-level, hard to read, hardware-bound
Assembly	Mnemonics (e.g., MOV, ADD)	One-to-one mapping to machine code
High-Level	C, Java, Python	Portable, readable, abstract
Translation	Compiler/Assembler	Bridge between human logic and hardware

What For?

Compiler design is essential to:

- Enable the execution of high-level code on physical machines.
- Detect errors during compilation time (before execution).
- Perform optimizations that enhance performance.
- Support program portability across platforms.

Language Processing System

A Language Processing System is a set of tools and phases that process a high-level program and convert it into machine code, ready for execution.

Language Types

Human Language: Natural languages like English; ambiguous and context-sensitive.

Programming Language: Structured, formal language used to write instructions for a computer.

Machine Language: Binary code (0s and 1s) directly understood by the CPU.

Components of a Language Processing System

1. Editor: Software used to write and edit source code. Examples include Notepad++, VS Code, etc.

2. Compiler: Translates high-level code into machine code or intermediate code. Performs lexical, syntax, semantic analysis, and code generation. Entire program is processed at once.

3. Interpreter: Executes the code line by line, translating and running instructions simultaneously.

Slower than compiler but useful for debugging.

Compiler

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

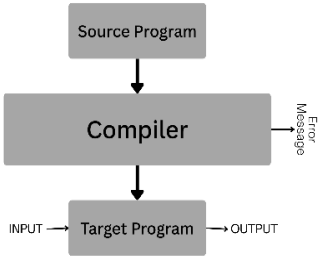


Fig: Basic Compiler

Interpreter

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

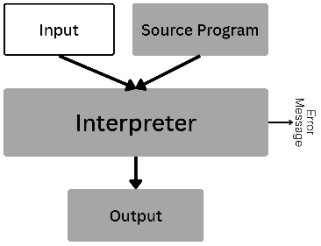


Fig: Basic Interpreter

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

Java language processors combine compilation and interpretation. A Java source program may first be compiled into an intermediate form called bytecodes. The bytecodes are then interpreted by a virtual machine. A benefit of this arrangement is that bytecodes compiled on one machine can be interpreted on another machine, perhaps across a network.

Difference between Compiler and Interpreter:

Feature	Compiler	Interpreter
Translation	Translates full code at once	Translates line by line
Execution Speed	Fast	Slower
Output	Generates machine code	No separate output file
Error Handling	Shows all errors at once	Stops at first error
Use Case	Production-level programs	Testing, scripting

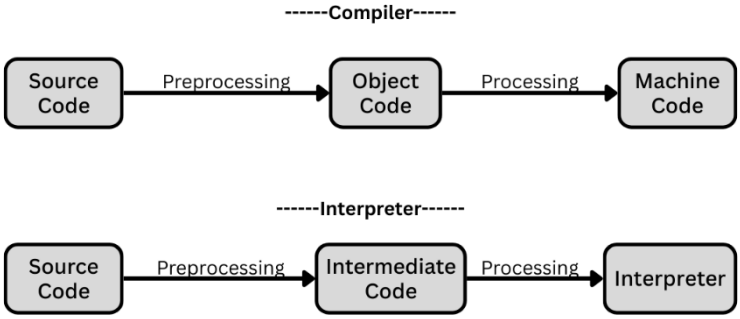


Fig: Compiler & Interpreter

Steps of Language Processing System

We know a computer is a logical assembly of Software and Hardware. The hardware knows a language, that is hard for us to understand, consequently we tend to write programs in high-level language, that is much less complicated for us to comprehend and maintain in thoughts. Now these programs go through a series of transformation so that they can readily be used in machines. This is where language procedure systems come handy.

- High Level Language– If a program contains #define or #include directives such as #include or #define it is called HLL. They are closer to humans but far from machines. These (#) tags are called pre-processor directives. They direct the pre-processor about what to do.
- Pre-Processor– The pre-processor removes all the #include directives by including the files called file inclusion and all the #define directives using macro expansion. It performs file inclusion, augmentation, macro-processing etc.
- Assembly Language– Its neither in binary form nor high level. It is an intermediate state that is a combination of machine instructions and some other useful data needed for execution.
- Assembler– For every platform (Hardware+ OS) we will have an assembler. They are not universal since for each platform we have one. The output of assembler is called object file. It translates assembly language to machine code.
- Interpreter– An interpreter converts high level language into low level machine language, just like a compiler. But they are different in the way they read the input. The Compiler in one go reads the inputs, does the processing and executes the source code whereas the interpreter does the same line by line. Compiler scans the entire program and translates it as a whole into machine code whereas an interpreter translates the program one statement at a time. Interpreted programs are usually slower with respect to compiled ones.

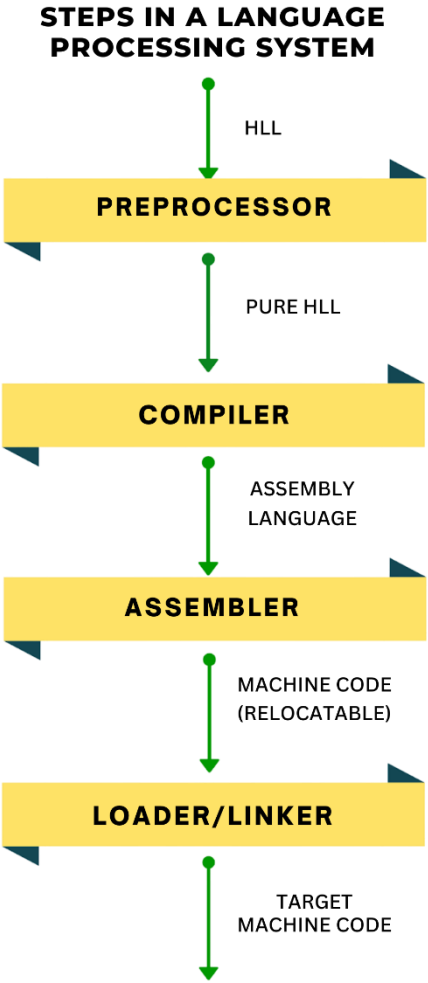


Fig: Steps in Language Processing System

- Relocatable Machine Code– It can be loaded at any point and can be run. The address within the program will be in such a way that it will cooperate for the program movement.
- Loader/Linker– It converts the relocatable code into absolute code and tries to run the program resulting in a running program or an error message (or sometimes both can happen). Linker loads a variety of object files into a single file to make it executable. Then loader loads it in memory and executes it.

Tokens, Lexemes, and Patterns

Token

A token is a category or class of lexemes. It represents a group of similar strings in a language, such as identifiers, keywords, operators, constants, etc.
Examples of Tokens:

Operators	=, +, –, (, {, :, =, ==, <, >
Keywords	if, while, for, int, double
Numeric literals	43, 6.035, -3.6e10, 0x13F3A
Character literals	'a', '~', '\'
String literals	"3.142", "aBcDe", "\"

Examples of non-tokens:

White space	space(' '), tab('\t'), eoln('\n')
Comments	1. /*this is not a token*/

Lexeme

A lexeme is the actual sequence of characters in the source code that matches the pattern for a token.
Example:
In the statement, **int count = 10;**
int is a lexeme for the token KEYWORD
count is a lexeme for the token IDENTIFIER
10 is a lexeme for the token CONSTANT

Pattern

A pattern is a rule or description that defines a set of strings. It is used by the lexical analyzer to identify lexemes and assign them to appropriate tokens.

Example:
Pattern for an identifier: letter (letter | digit)*

These concepts are crucial in lexical analysis—the first phase of a compiler—where the source code is tokenized for further syntactic processing.

Practice Questions

1. Explain the role of a compiler in language processing.
2. Differentiate between compiler and interpreter.
3. Describe the importance of language processing systems.
4. Write short notes on assembler, linker, and loader.
5. Illustrate the components of the language processing system with a diagram.

Viva Questions

1. What is a compiler?
2. Define lexeme and token.
3. What is the difference between syntax and semantics?
4. What is an interpreter?
5. Name some language processing tools.

MCQs

1. Which of the following is the main function of a compiler?
A) Execute programs
B) Translate high-level code into machine code
C) Manage memory
D) Debug code
Answer: B

2

Phases of Compiler

The structure of a compiler is broadly divided into two major parts:
1. Analysis / Front-end phase
2. Synthesis/Back-end phase

Analysis Phase

This phase reads the source program and breaks it into meaningful components. It focuses on understanding the code and extracting information from it. The analysis phase is also called the front-end of the compiler.

Synthesis Phase

This phase takes the intermediate representation generated by the analysis phase and transforms it into an executable form. It focuses on generating the correct and optimized machine code. The synthesis phase is known as the back-end of the compiler.

The process of compilation is divided into several well-defined phases, each responsible for a specific task in translating source code into executable machine code. These phases work together in a sequence to analyze, transform, and generate code. The major phases include:

- Lexical Analysis, where the input is broken into tokens.
- Syntax Analysis, which checks the grammatical structure.
- Semantic Analysis, ensuring semantic correctness.
- Intermediate Code Generation, converting source to an intermediate representation.
- Code Optimization, improving performance without changing functionality.

- Code Generation, producing target machine code.
- Symbol Table Management and Error Handling are supporting activities that occur throughout the process.

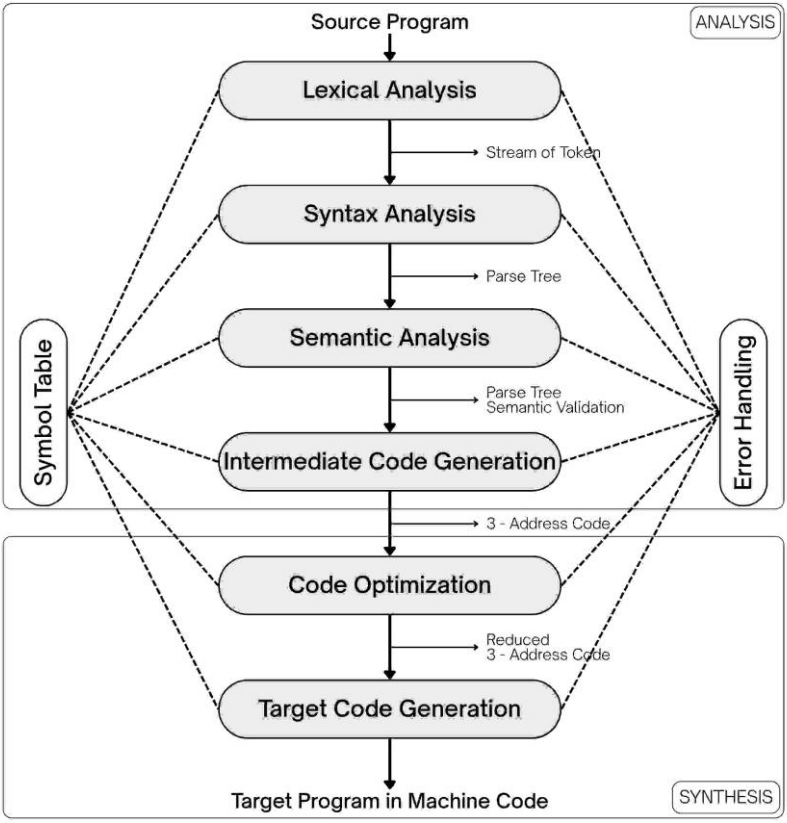


Fig: Phases of Compiler

Lexical Analysis Phase

The first phase of a compiler is called lexical analysis or scanning. The lexical analyzer reads the stream of characters making up the source program and groups the characters into meaningful sequences called lexemes. For each lexeme, the lexical analyzer produces as output a token of the form:

{token- name, attribute-value}

That it passes on to the subsequent phase, syntax analysis. In the token, the first component token- name is an abstract symbol that is used during syntax analysis, and the second component attribute-value points to an entry in the symbol table for this token. Information from the symbol- table entry 'is needed for semantic analysis and code generation.

- Converts a stream of characters into a stream of tokens.
- Removes white spaces and comments.
- Recognizes lexemes (basic syntactic units).
- Tokens: keywords, identifiers, literals, operators, punctuations.

Example:

Equation:

Position = Initial + Rate * 60

Symbol Table:

Address	Symbol	Type	Attributes
1	Position	int	<id,1>
2	Initial	int	<id,2>
3	Rate	Int	<id,3>
4	60	Const	<id,4>
5	=	Operator	<op, 5>
6	+	Operator	<op, 6>
7	*	Operator	<op, 7>

Lexical Form:

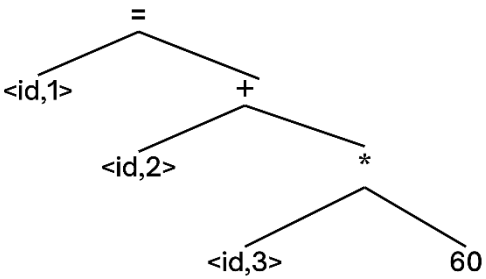
<id,1> = <id,2> + <id,3> * 60

Syntax Analysis Phase

The second phase of the compiler is syntax analysis or parsing. The parser uses the first components of the tokens produced by the lexical analyzer to create a tree-like intermediate representation that depicts the grammatical structure of the token stream. A typical representation is a syntax tree in which each interior node represents an operation and the children of the node represent the arguments of the operation.

- Takes the token stream and checks it against grammatical rules defined in a Context-Free Grammar (CFG).
- Produces a Parse Tree or Syntax Tree.
- Detects: Misuse of syntax
- Detects: Improper structure (e.g., missing braces, wrong nesting)

Syntax Tree (parse tree):



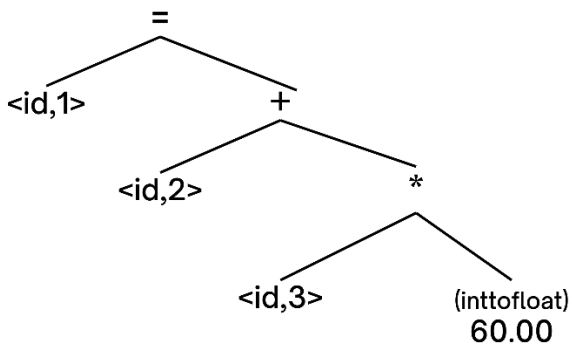
Semantic Analysis Phase

The semantic analyzer uses the syntax tree and the information in the symbol table to check the source program for semantic consistency with the language definition. Gathers type information and saves it in either the syntax tree or the symbol table, for subsequent use during intermediate-code generation.

An important part of semantic analysis is type checking, where the compiler checks that each operator has matching operands. For example, many programming language definitions require an array index to be an integer; the compiler must report an error if a floating-point number is used to index an array.

- Checks the semantic validity of the parse tree.
- Validates types, scopes, and declarations.
- Uses symbol table to store variable/function information.
- Detects: Undeclared variables
- Detects: Type mismatches (e.g., assigning float to int without casting)

Semantic Tree:



Intermediate Code Generation Phase

After syntax and semantic analysis of the source program, many compilers generate an explicit low-level or machine-like intermediate representation (a program for an abstract machine). This intermediate representation should have two important properties:

- It should be easy to produce and
- It should be easy to translate into the target machine.
-

The considered intermediate form called three-address code, which consists of a sequence of assembly-like instructions with three operands per instruction. Each operand can act like a register.

This phase bridges the analysis and synthesis phases of translation.

- Generates an Intermediate Representation (IR) that is independent of both the source and target languages.
- Simplifies optimization and code generation.

Intermediate Code:

t1 = intofloat(60.00)

t2 = <id,3> * t1

t3 = <id,2> + t2

<id,1> = t3

Code Optimization Phase

The compiler looks at large segments of the program to decide how to improve performance. The machine-independent code-optimization phase attempts to improve the intermediate code so that better target code will result. Usually better means: faster, shorter code, or target code that consumes less power. There are simple optimizations that significantly improve the running time of the target program without slowing down compilation too much. Optimization cannot make an inefficient algorithm efficient - "only makes an efficient algorithm more efficient"

- Improves the quality of intermediate code for performance.
- Maintains semantic equivalence while reducing resource usage.

Optimized Code:

t1 = <id,3> * 60.00

<id,1> = <id,2> + t1

Code Generation Phase

The last phase of translation is code generation. Takes as input an intermediate representation of the source program and maps it into the target language. If the target language is machine, code, registers or memory locations are selected for each of the variables used by the program. Then, the intermediate instructions are translated into sequences of machine instructions that perform the same task. A crucial aspect of code generation is the judicious assignment of registers to hold variables.

- Translates the optimized IR into target machine code.
- Handles register allocation, instruction selection, and addressing modes.

Assembly Language Code:

```
LDF     R2,  <id,3>
MULF    R2,  R2,  #60.00
LDF     R1,  <id,2>
ADDF    R1,  R1,  R2
STF     <id,1>,   R1
```

Symbol-Table Management

An essential function of a compiler is to record the variable names used in the source program and collect information about various attributes of each name. These attributes may provide information about the storage allocated for a name, its type, its scope (where in the program its value may be used), and in the case of procedure names, such things as the number and types of its arguments, the method of passing each argument (for example, by value or by reference), and the type returned.

The symbol table is a data structure containing a record for each variable name, with fields for the attributes of the name. The data structure should be designed to allow the compiler to find the record for each name quickly and to store or retrieve data from that record quickly.

Summary of Compiler Phases

Phase	Input	Output	Belongs To
Lexical Analysis	Source Code	Tokens	Analysis
Syntax Analysis	Tokens	Parse Tree	Analysis
Semantic Analysis	Parse Tree	Annotated Tree	Analysis
Intermediate Code	Annotated Tree	IR (TAC, AST)	Synthesis
Optimization	IR	Optimized IR	Synthesis
Code Generation	Optimized IR	Target Code	Synthesis

Compiler- Construction Tools

1. Parser generators that automatically produce syntax analyzers from a grammatical description of a programming language.
2. Scanner generators that produce lexical analyzers from a regular expression description of the tokens of a language.
3. Syntax-directed translation engines that produce collections of routines for walking a parse tree and generating intermediate code.
4. Code-generator generators that produce a code generator from a collection of rules for translating each operation of the intermediate language into the machine language for a target machine.
5. Data-flow analysis engines that facilitate the gathering of information about how values are transmitted from one part of a program to each 1other part. Data-flow analysis is a key part of code optimization.
6. Compiler- construction toolkits that provide an integrated set of routines for constructing various phases of a compiler.

Errors in Compilation

During compilation, errors may arise at different phases. Identifying and handling these errors is a vital part of compiler design.
Here are the major types of errors:

Lexical Errors: Errors due to invalid tokens (e.g., illegal characters in identifiers).

Syntax Errors: Violations of grammatical rules of the language (detected during parsing).

Semantic Errors: Errors due to meaning-related issues, such as incompatible data types or undeclared variables.

Logical Errors: The program runs but produces incorrect results due to a flaw in logic.

Runtime Errors: Errors that occur during program execution, such as division by zero.

Example:

```
int main() {
    int a, b;
    intt result;   // Lexical Error: Invalid token ‘intt’
    a = 10;
    b = 0;
    result = a + ;  // Syntax Error: Missing operand after '+'
    int x = "hello"; // Semantic Error: Incompatible type assignment (string
to int)
    result = a - b; // Correct
    result = a / b; // Semantic Error: Division by zero (may also cause
Runtime Error)
    result = a * b;   // Logical Error: Suppose user intended to add but
mistakenly multiplied
    printf("Result is: %d\n", result);
    return 0;
}
```

Compilers are designed to detect and recover from as many errors as possible to help the programmer debug code efficiently.

Practice Questions

1. Explain the six phases of a compiler.
2. Describe the analysis and synthesis parts of a compiler.
3. What are the responsibilities of the lexical analysis phase?
4. Differentiate between intermediate code generation and code optimization.
5. Explain how errors are handled during compilation.

Viva Questions

1. Name the phases of a compiler.
2. What is lexical analysis?
3. What is the purpose of the semantic analysis phase?
4. What kind of errors occur during compilation?
5. What is a compiler-construction tool?

3

Context Free Grammer (CFG)

A Context-Free Grammar (CFG) is a formal grammar used to describe the syntax of programming languages. It is essential in the design of compilers, especially in the syntax analysis phase, where the structure of source code is verified according to a defined grammar.

A Context-Free Grammar consists of four components:
G = (V, T, P, S)
Where:
N → Finite set of variables (also called non-terminals)
T → Finite set of terminals (actual symbols used in the language)
P → Finite set of production rules, each of the form A → α
S → Start symbol, a special variable from V that begins the derivation
A → α means that variable A can be replaced by string α, where α is a combination of terminals and/or variables.

Properties of Context Free Grammar

- All Regular Languages along with some Non-regular Languages are Context Free Languages. Context Free Language is a language based on Context Free Grammar. Therefore, all Regular Languages = Context Free Languages. Some / not all Non-regular Languages = Context Free Languages.

- If a Language L is not a Context Free Language, then the language L is a Non-regular Language.
- All Regular Languages along with some Non-regular Languages are Context Free Languages. Context Free Language is a language based on Context Free Grammar. Therefore, all Regular Languages = Context Free Languages. Some / not all Non-regular Languages = Context Free Languages.

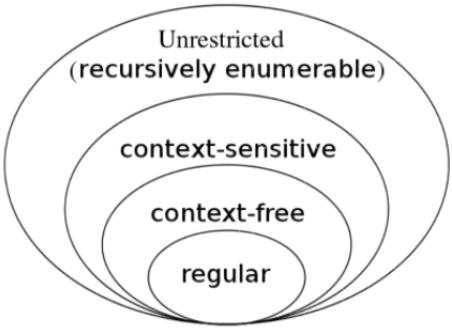


Fig: Grammer in CD

- If a Language L is not a Context Free Language, then the language L is a Non-regular Language.

CFG-Derivation

Derivations in CFG

A derivation is basically a sequence of production rules, in order to get the input string. During parsing, we take two decisions for some sentential form of input:
• Deciding the non-terminal which is to be replaced.
• Deciding the production rule, by which, the non-terminal will be replaced.
• To decide which non-terminal to be replaced with production rule, we can have two options.

Types of Derivation

1. **Left-most Derivation (LMD):** If the sentential form of an input is scanned and replaced from left to right, it is called left-most derivation. The sentential form derived by the left-most derivation is called the left-sentential form.

2. **Right-most Derivation (RMD):** If we scan and replace the input with production rules, from right to left, it is known as right-most derivation. The sentential form derived from the right-most derivation is called the right-sentential form.

Example:

Production rules:

$E \rightarrow E + E$

$E \rightarrow E * E$

$E \rightarrow id$

Input string: id + id * id

So, LMD and RMD derivation will be:

LMD	RMD
$E \rightarrow E * E$	$E \rightarrow E + E$
$E \rightarrow E + E * E$	$E \rightarrow E + E * E$
$E \rightarrow id + E * E$	$E \rightarrow E + E * id$
$E \rightarrow id + id * E$	$E \rightarrow E + id * id$
$E \rightarrow id + id * id$	$E \rightarrow id + id * id$

Parse Tree

A parse tree is a hierarchical, tree-like representation of the derivation of a string according to a context-free grammar. It provides a visual structure of how a string is generated from the grammar's start symbol, showing each step of the derivation clearly.

- It is a graphical depiction of a derivation.
- It helps visualize how strings are derived from the start symbol.

The root of the parse tree is the grammar's start symbol, and the leaves represent the terminals of the derived string.

Each internal node of the tree corresponds to a grammar symbol, and the children of a node represent the production used to expand that symbol. The tree illustrates not just the syntactic structure but also helps in analyzing ambiguity, precedence, and associativity.

For every parse tree, there exists:

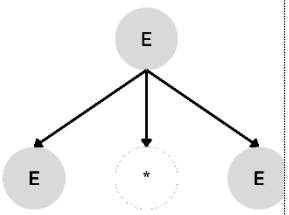
- A unique leftmost d unique leftmost derivation (where the leftmost non-terminal is expanded first at each step).

- A unique rightmost derivation (where the rightmost non-terminal is expanded first).

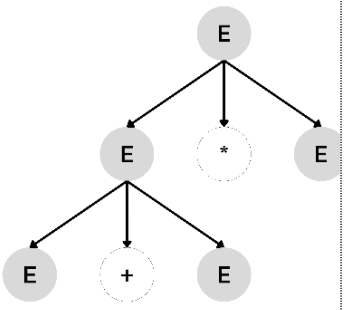
We categorize parser into two groups:
1. Top-Down Parser
2. Bottom-Up Parser.

Constructing the Parse Tree

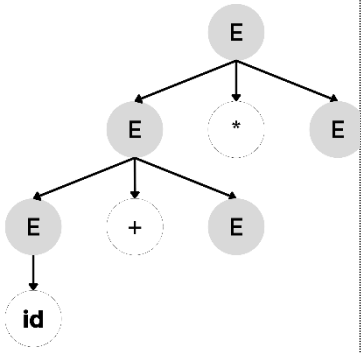
Step-1:
E → E * E



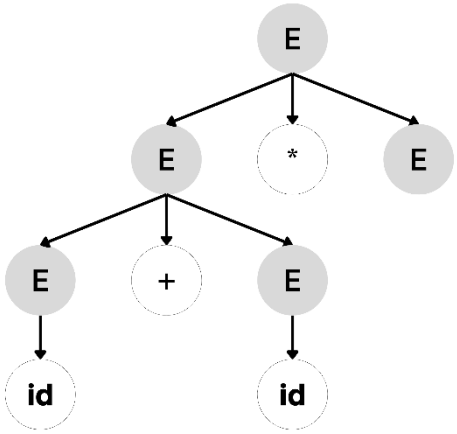
Step-2:
E → E + E * E



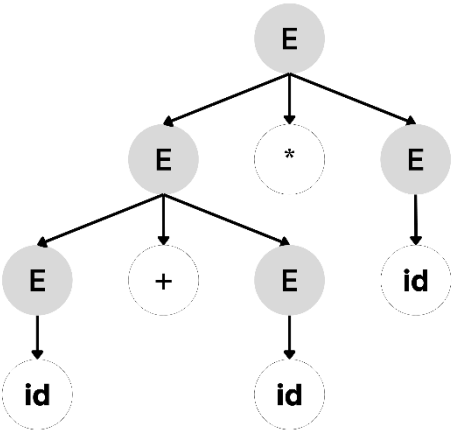
Step-3:
E → id + E * E



Step-4:
E → id + id * E



Step-5:
E → id + id * id



Ambiguity

A grammar is ambiguous if there exists a string with more than one parse tree or more than one left/right derivation.

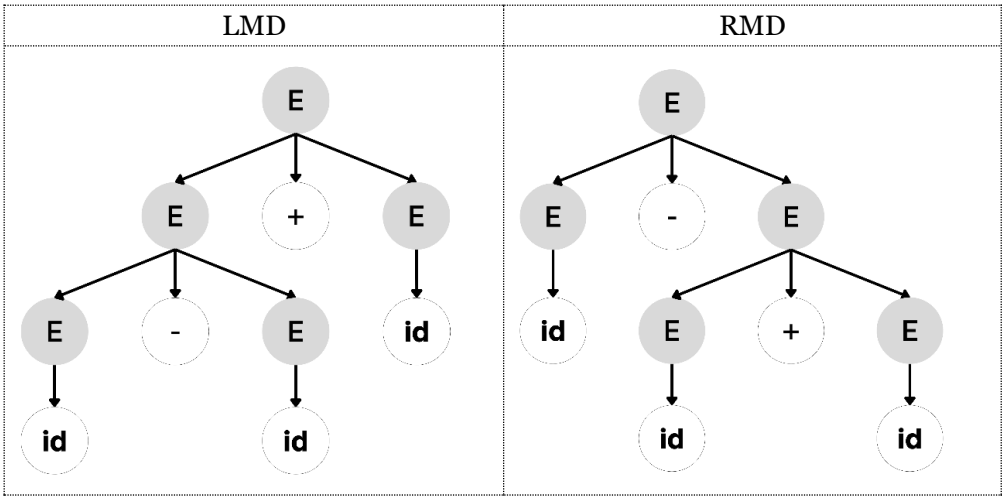
Example:

$E \rightarrow E + E$

$E \rightarrow E - E$

$E \rightarrow id$

For the string id + id – id, the above grammar generates two parse trees:



Top-Down Parsing

Parsing is the process of determining if a string of tokens can be generated by a grammar.

For any context-free grammar there is a parser that takes at most O(n3) time to parse a string of n tokens.

- Top-down parsers build parse trees from the top (root) to the bottom (leaves).
- Two top-down parsing are to be discussed:
  - Recursive Descent Parsing
  - Predictive Parsing An efficient non-backtracking parsing called for LL(1) grammars.

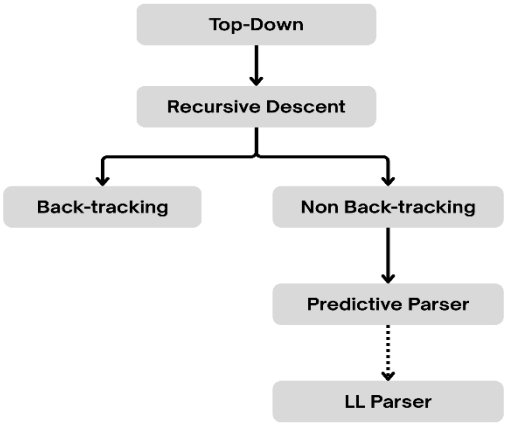


Fig: Phases of Compiler

Example of Top-Down Parsing

Consider the grammar:

$S \rightarrow cAd$

$A \rightarrow ab \mid a$

Input string: w = cad

To construct a parse tree for this string using top-down approach, initially create a tree consisting of a single node labeled S.

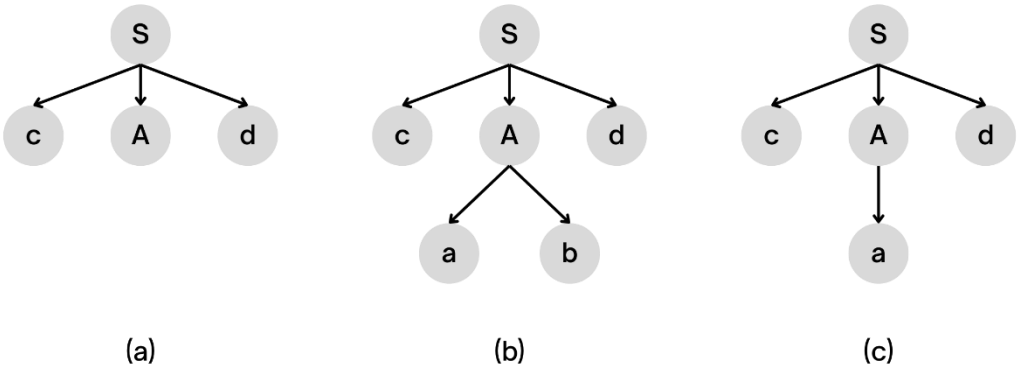


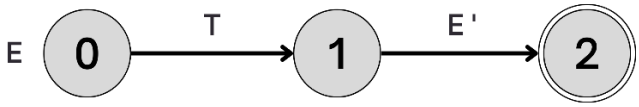
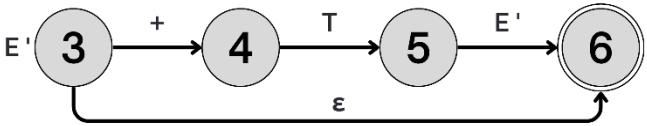
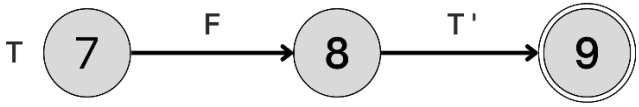
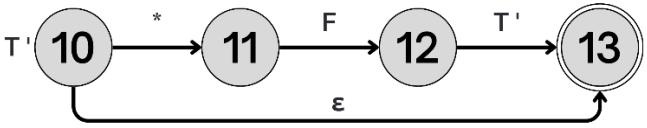
Fig: Steps in Top-Down Parse

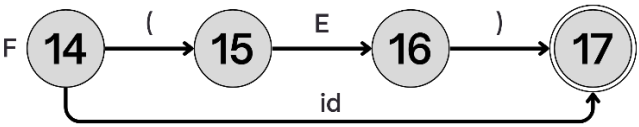
Procedure of Top-down Parsing

- An input pointer points to c, the first symbol of w.
- Then use the first production for S to expand the tree and obtain the tree.

- The leftmost leaf, labeled c, matches the first symbol of w.
- Next input pointer to a, the second symbol of w.
- Consider the next leaf, labeled A.
- Expand A using the first alternative for A to obtain the tree.
- Now have a match for the second input symbol. Then advance to the next input pointer d, the third input symbol and compare d against the next leaf, labeled b. Since b does not match d, report failure and go back to A to see whether there is another alternative. (Backtracking takes place).
- If there is another alternative for A, substitute and compare the input symbol with leaf.
- Repeat the step for all the alternatives of A to find a match using backtracking. If match found, then the string is accepted by the grammar. Else report failure.
- A left-recursive grammar can cause a recursive-descent parser, even one with backtracking, to go into an infinite loop.
- As discussed above, an easy way to implement a recursive descent parsing with backtracking is to create a procedure for each non-terminal.

Transition Diagram for Predictive Parsers

E → TE'	
E' → +TE' |ε	
T → FT'	
T' → * FT' | ε	

F → (E) | id	

Example:
Build the parse tree for the arithmetic expression 4 + 2 * 3 using the expression grammar:

E → E + T | E - T | T
T → T * F | F
F → a | (E)

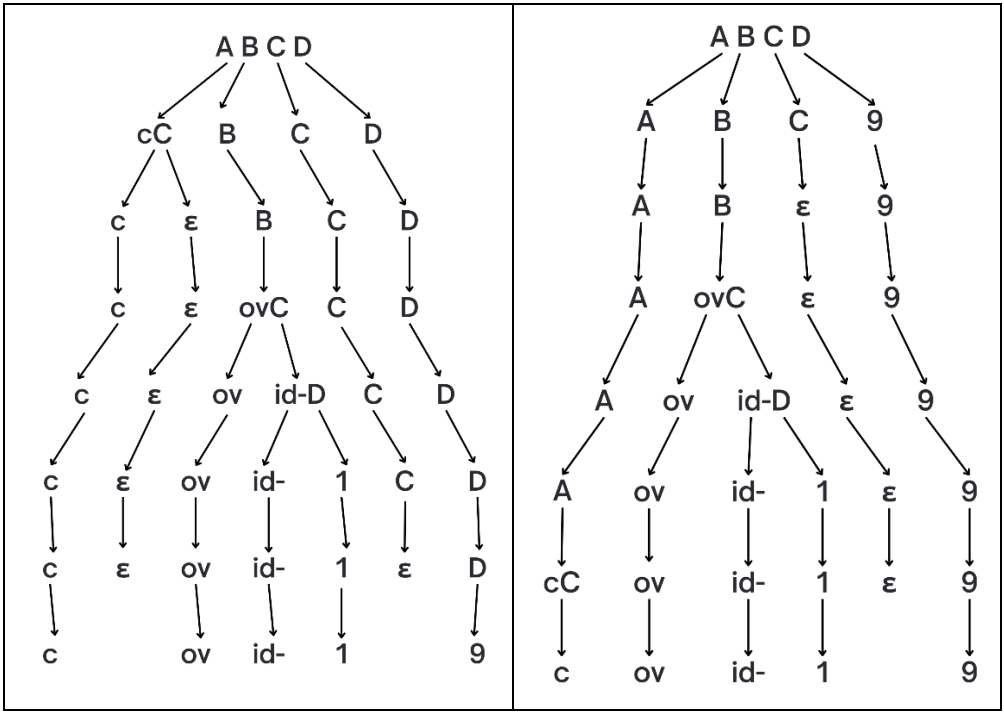
where a represents an operand of some type, be it a number or variable. The trees are grouped by height.

Example:
Given the following context-free grammar (CFG):
P → ABCD
A → aA | cC | oD | iC
B → ov | ovC
C → id | id-D | ε
D → 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

Tasks:
Show the LMD & RMD for the string covid-19. Based on your derivations, determine whether the grammar is ambiguous. Justify your answer.

Solution:
Leftmost Derivation (LMD):

Leftmost Derivation (LMD)	Rightmost Derivation (RMD)
P → ABCD → cC B C D → c ε B C D → c ε ovC C D → c ε ov id-D C D → c ε ov id- 1 C D → c ε ov id-1 ε D → c ov id-1 9	P → ABCD → A B C 9 → A B ε → A ovC ε 9 → A ov id-D ε 9 → A ov id- 1 ε 9 → cC ov id- 1 ε 9 → c ov id- 1 9



Ambiguity Demonstration:
If more than one parse tree or derivation exists for the same input string (like covid-19) using different choices of rules, the grammar is **ambiguous**.

Since the string covid-19 can be derived using more than one leftmost/rightmost derivation (with different parse trees), this CFG is ambiguous.

Practice Questions

1. Define CFG with a suitable example.
2. Derive a string using LMD and RMD.
3. Construct a parse tree for a given grammar.
4. Identify whether a grammar is ambiguous.
5. Convert a grammar into an equivalent unambiguous CFG.

Viva Questions

1. What is a context-free grammar?
2. Define derivation in CFG.
3. What is a parse tree?
4. What is ambiguity in grammar?
5. Can CFG represent all programming languages?

4

Regular Expression

A RegEx—Regular expression (sometimes called a rational expression) is a sequence of characters that define a search pattern, mainly for use in pattern matching with strings, or string matching, i.e. “find and replace”-like operations. Regular expression is a notation for defining the set of tokens that normally occur in programming languages.

Definition of Regular Expression

A regular expression is defined over an alphabet Σ and denotes a language. Formally, the rules are:

- ϕ represents the empty set.
- ε represents the empty string.
- a, for a ∈ Σ, represents a string with a single character "a".
- If r and s are regular expressions, then:
  - (r + s) represents union (either r or s),
  - (rs) represents concatenation,
  - (r)* represents Kleene star (zero or more repetitions of r).

Regular expressions play a vital role in the lexical analysis phase of a compiler, where they are used to describe the patterns of valid tokens such as identifiers, keywords, operators, and literals. These patterns are then converted into finite automata, which serve as the basis for implementing lexical analyzers. Understanding regular expressions is essential not only for compiler construction but also for text processing, data validation, and many areas of software development. Mastery of regular expressions allows a programmer to succinctly define and detect complex text patterns with precision and efficiency.

Regular Expression Operators

Operator	Name	Meaning	Example
+	Union/OR	Matches either pattern	a + b → a or b
.	Concatenation	Matches one after another	ab → ab
*	Kleene Star	Matches zero or more occurrences	a* → "", "a", "aa", etc.
?	Optional	Matches zero or one occurrence	a? → "", "a"
[]	Character set	Matches any one character inside	[abc] → a, b, or c
()	Grouping	Used for precedence and subpatterns	(ab)*

Regular Expressions to Language

Examples:

Let L = {a, b}

Some regular expressions:

1. a | b

⇨ Denotes the set of {a, b} having a or b.

2. (a | b) (a | b)

⇨ Denotes {aa, ab, ba, bb}, the set of all strings of a's and b's of length two.

3. a*

⇨ Denotes the set of all strings of zero or more a's , i. e., {ε, a, aa, aaa, ...}

4. L^4

⇨ Denotes the set of all four-letter strings.

5. (a | b)* or (a* | b*)*

⇨ Denotes the set of all strings containing zero or more instances of an a or b, that is, the set of all strings of a's and b's.

6. a | a* b

⇨ Denotes the set containing the string a and all strings consisting of zero or more a's followed by a b.

Language to Regular Expressions

Examples:

1. "Set of all strings having at least one ab"

⇨ (ab)+

2. "Set of all strings having even number of aa"

⇨ (aa)*

3. "Set of all strings having odd number of bb"

⇨ b(bb)*

4. "Set of all strings having even number of aa and even number of bb"

⇨ (aa)* (bb)*

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

⇨ (aa)(a | b)*

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

⇨ (a | b)* (bb)

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

⇨ (aa) (a | b)* (bb)

Regular Expressions vs Regular Languages

Regular Expressions	Regular Languages
Describe patterns in strings	Set of strings over an alphabet
Define regular languages	Described by regular expressions
Basis for building finite automata	Recognized by finite automata (NFA/DFA)

Practice Questions

Write a regular expression for:

1. All strings containing only a's and b's with at least one a.
2. All valid decimal numbers.
3. All identifiers starting with a letter and followed by letters or digits.
4. Convert the regular expression (a+b)*abb into an NFA.
5. Prove whether the language represented by (aa+bb)* is regular.

Viva Questions

1. What is a regular expression?
2. Name the operators used in regular expressions.
3. What are regular languages?
4. Can regular expressions handle nested structures?
5. Give an example of a regular expression for an email.

5

NFA & DFA

NFA → Non-Deterministic Finite Automata

DFA → Deterministic Finite Automata

Introduction

The term "Automata" is derived from the Greek word "αὐτόματα" which means "self-acting". An automaton (Automata in plural) is an abstract self-propelled computing device which follows a predetermined sequence of operations automatically.

An automaton with a finite number of states is called a Finite Automaton (FA) or Finite State Machine (FSM).

In the process of lexical analysis, we need to recognize whether a given string belongs to a particular language defined by a regular expression. To do this efficiently, we use finite automata. There are two types:

1. Deterministic Finite Automaton (DFA)
2. Non-deterministic Finite Automaton (NFA)

Both are used to recognize regular languages, and each has a formal structure and specific working behavior.

Finite Automata Basics

An automaton can be represented by a 5-tuple $(Q, \sum, \delta, q_o, F)$, where –

$Q \rightarrow$ A finite set of states
$\sum \rightarrow$ A finite set of input symbols
$\delta \rightarrow$ A transition function
$q_o \rightarrow$ A start state ($q_o \in Q$)
$F \rightarrow$ A set of accept states ($F \subseteq Q$)

Related Terminologies

Alphabet

Definition: An alphabet is any finite set of symbols.
Example: $\sum$ = {a, b, c, d} is an alphabet set where 'a', 'b', 'c', and 'd' are symbols.

String

Definition: A string is a finite sequence of symbols taken from $\sum$.
Example: 'cabcad' is a valid string on the alphabet set $\sum$ = {a, b, c, d}

Length of a String

Definition: It is the number of symbols present in a string. (Denoted by |S|).
Examples: If S = 'cabcad', |S|= 6
If |S|= 0, it is called an empty string (Denoted by λ or ε)

Kleene Star

Definition: The Kleene star, $\sum^*$, is a unary operator on a set of symbols or strings, $\sum$, that gives the infinite set of all possible strings of all possible lengths over $\sum$ including λ.
Representation: $\sum^* = \sum 0 \cup \sum 1 \cup \sum 2 \cup \ldots\ldots$ where $\sum p$ is the set of all possible strings of length p.
Example: If $\sum$ = {a, b}, $\sum^*$ = {λ, a, b, aa, ab, ba, bb,.....}

Kleene Closure / Plus

Definition: The set $\sum^+$ is the infinite set of all possible strings of all possible lengths over $\sum$ excluding λ.
Representation: $\sum^+ = \sum 1 \cup \sum 2 \cup \sum 3 \cup \ldots\ldots$

$$\sum^+ = \sum^* - \{\lambda\}$$

Example: If $\sum = \{ a, b \}$,

$\sum^+ = \{ a, b, aa, ab, ba, bb,.....\}$

Language

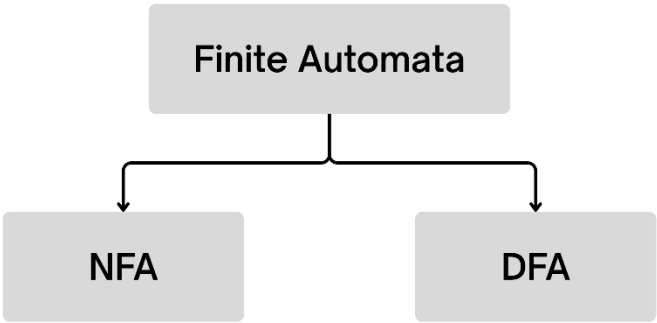
Definition: A language is a subset of $\sum^*$ for some alphabet $\sum$. It can be finite or infinite.

Example: If the language takes all possible strings of length 2 over $\sum = \{a, b\}$, then $L = \{ ab, aa, ba, bb \}$

Types of Finite Automaton

Finite Automaton can be classified into two types –

1. Deterministic Finite Automaton (DFA)
2. Non-deterministic Finite Automaton (NDFA / NFA)



Non-Deterministic Finite Automaton (NFA)

In NFA, for a given state and input symbol, there can be multiple transitions. It also allows ε-transitions (moves without input).

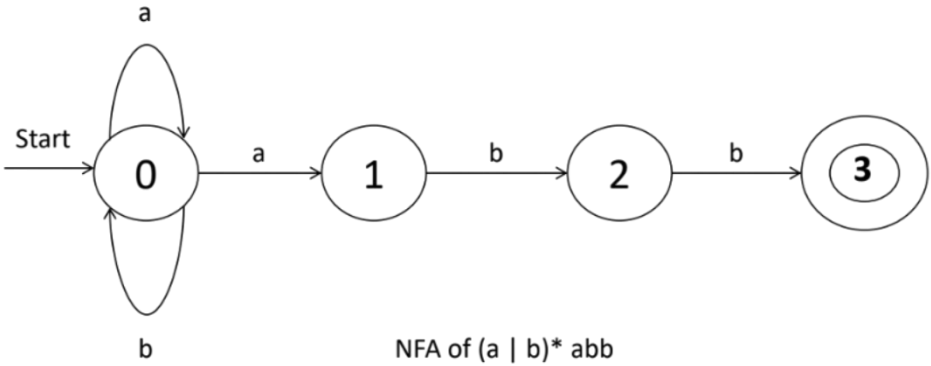
Graphical Representation of a NFA

An NFA is represented by digraphs called state diagram.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

Example

Regular Expression: (a | b)* abb



NFA of (a | b)* abb

Let's find the 5 tuples:

1. States: {0, 1, 2, 3}
2. Input Symbol: {a, b}
3. Start State: {0}
4. Final State: {3}
5. Transition function:

Transition table shows the function

State	Input Symbol	
	a	b
0	{0,1}	{0}
1	Ø	{2}
2	Ø	{3}

Deterministic Finite Automaton (DFA)

In DFA, for every state and input symbol, there is exactly one transition.

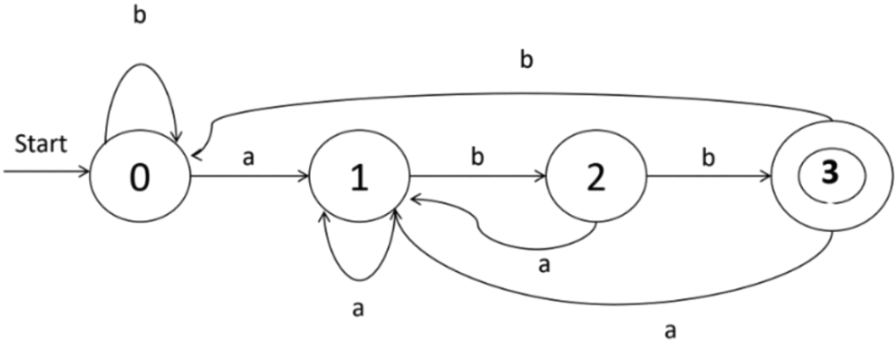
Graphical Representation of a NFA

A DFA is represented by digraphs called state diagram.

- The vertices represent the states.
- The arcs labeled with an input alphabet show the transitions.
- The initial state is denoted by an empty single incoming arc.
- The final state is indicated by double circles.

Example

Regular Expression: (a | b)* abb



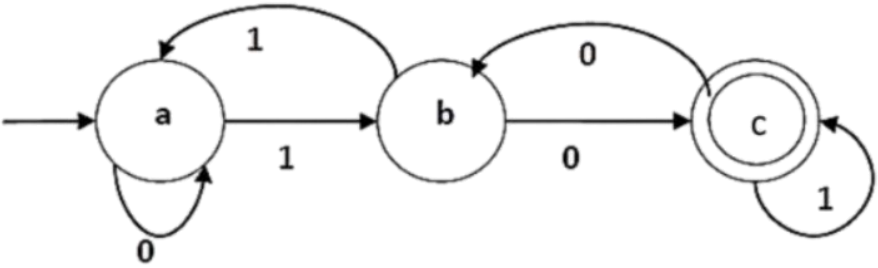
Let a deterministic finite automaton be –

1. Q = {a, b, c},
2. ∑ = {0, 1},
3. q_o = {a},
4. F = {c},
5. Transition function:

Transition function δ as shown by the following table –

Present State	Next State for Input 0	Next State for Input 1
a	a	b
b	c	a
c	b	c

Its graphical representation would be as follows –



Differences between DFA and NFA:

DFA	NFA
The transition from a state is to a single particular next state for each input symbol. Hence it is called deterministic.	The transition from a state can be to multiple next states for each input symbol. Hence it is called non-deterministic.
Empty string transitions are not seen in DFA.	NFA permits empty string transitions.
Backtracking is allowed in DFA	In NFA, backtracking is not always possible.
Requires more space.	Requires less space.
A string is accepted by a DFA, if it transits to a final state.	A string is accepted by a NFA, if at least one of all possible transitions ends in a final state.

NFA to DFA

Non-Deterministic Features of NFA

- There are three main cases of non- determinism in NFAs:
- Transition to a state without consuming any input.
- Multiple transitions on the same input symbol.
- No transition on an input symbol.

To convert NFAs to DFAs we need to get rid of non-determinism from NFAs.

Subset Construction Method

Using Subset construction method to convert NFA to DFA
involves the following steps:

1. For every state in the NFA, determine all reachable states for every input symbol.
2. The set of reachable states constitute a single state in the converted DFA (Each state in the DFA corresponds to a subset of states in the NFA).
3. Find reachable states for each new DFA state, until no more new states can be found.

Example

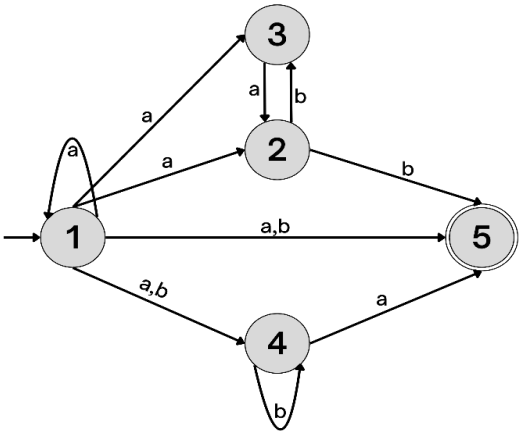


Fig: NFA without λ-transitions

Table: Transition table

q	a	b
→1	{1,2,3,4,5}	{4,5}
2	{3}	{5}
3	∅	{2}
4	{5}	{4}
5	∅	∅

Subset Construction Table:

q	a	b
1	{1,2,3,4,5}	{4,5}
{1,2,3,4,5}	{1,2,3,4,5}	{2,4,5}
{4,5}	5	4
{2,4,5}	{3,5}	{4,5}
5	∅	∅
4	5	4
{3,5}	∅	3
∅	∅	∅
2	3	5
3	∅	2

DFA from Subset Table:

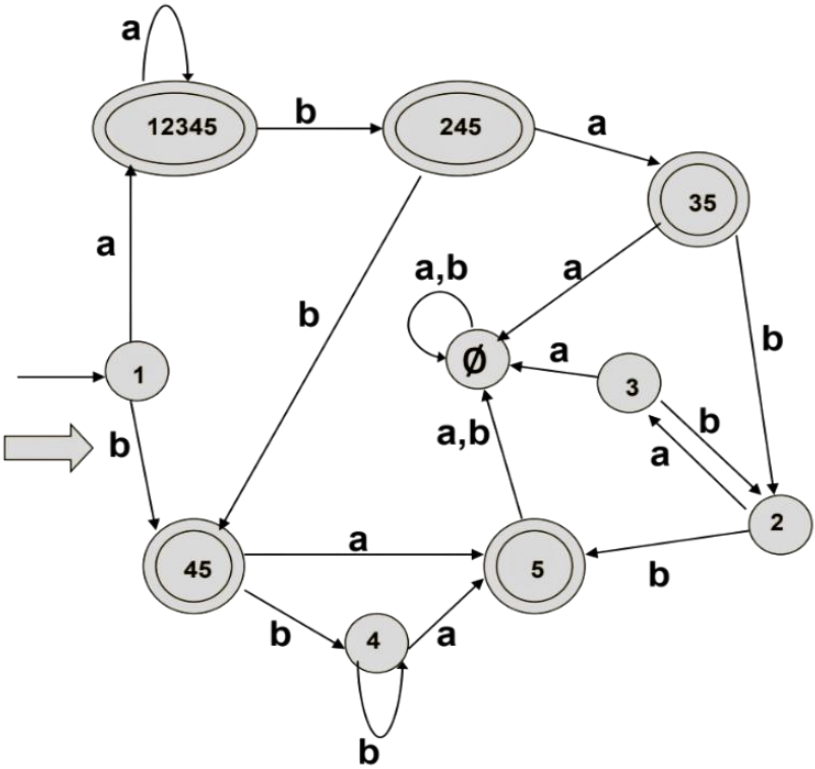


Fig: Final DFA graph from Constructed Subset Table

Practice Questions

1. Design a DFA to recognize binary numbers divisible by 3.
2. Convert an NFA to DFA with a step-by-step explanation.
3. Prove a given DFA accepts a specific language.
4. Show the difference between DFA and NFA with diagrams.
5. Explain the ε-closure concept in NFA.

Viva Questions

1. What is a DFA?

2. Define NFA.
3. What is the difference between DFA and NFA?
4. Is every NFA convertible to DFA?
5. What is ε-transition?

6

Left Recursion & Left Factoring

In the context of context-free grammars (CFG), some grammars can lead to parsing difficulties due to their structure. Two such issues are:

- Left Recursion — which causes problems for top-down parsers.
- Left Factoring — which is used to make grammars suitable for predictive parsing.

Both are crucial transformations to prepare grammars for use in parsers, particularly LL(1) parsers.

Left and Right Recursive Grammars

In a context-free grammar G, if there is a production in the form $X \rightarrow Xa$ where X is a non-terminal and 'a' is a string of terminals, it is called a left recursive production. The grammar having a left recursive production is called a left recursive grammar.

And if in a context-free grammar G, if there is a production is in the form $X \rightarrow aX$ where X is a non-terminal and 'a' is a string of terminals, it is called a right recursive production. The grammar having a right recursive production is called a right recursive grammar.

Left recursion often causes issues in top-down parsers like recursive descent, as it can lead to infinite recursion. Right recursion, on the other hand, is generally more parser-friendly and commonly used in such parsing techniques.

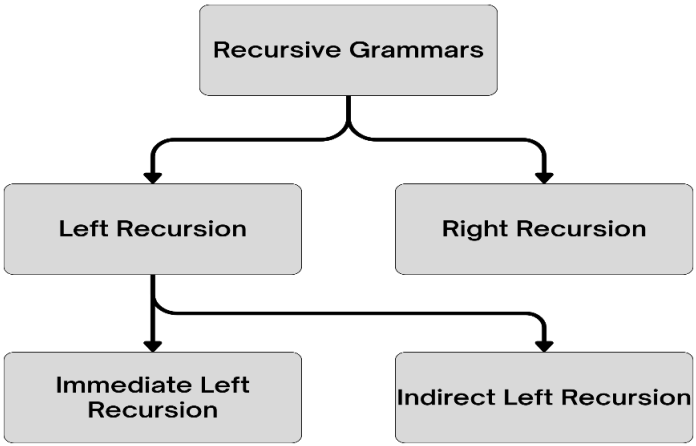
Left Recursion

A grammar is left recursive if there exists a non-terminal A such that:

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

This leads to an infinite recursion in top-down parsers like recursive descent.

Types of Left Recursion



1. Immediate Left Recursion: Occurs within a single production.

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

2. Indirect Left Recursion: Involves multiple non-terminals.

$$A \rightarrow B\alpha$$

$$B \rightarrow A\beta$$

Elimination of left recursion:

Left recursion is eliminated by converting the grammar into a right recursive grammar.

If we have the left-recursive pair of productions: $A \rightarrow A\alpha \mid \beta$, where β does not begin with an A. Then, we can eliminate left recursion by replacing the pair of productions with:

$$A \rightarrow \beta A'$$

$$A' \rightarrow \alpha A' \mid \varepsilon$$

This right recursive grammar functions same as left recursive grammar.

Example:

1. Consider the following grammar and eliminate left recursion:

$$A \rightarrow ABd \mid Aa \mid a$$
$$B \rightarrow Be \mid b$$

Solution:

The grammar after eliminating left recursion is:

$A \rightarrow aA'$

$A' \rightarrow BdA' / aA' / \varepsilon$

$B \rightarrow bB'$

$B' \rightarrow eB' / \varepsilon$

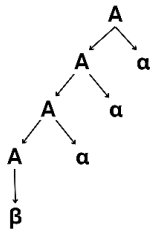
2. Consider the following grammar and eliminate left recursion:

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

Solution:

$A \rightarrow \beta\, A'$

$A' \rightarrow \varepsilon \mid \alpha\, A'$



$E \rightarrow T\, E'$

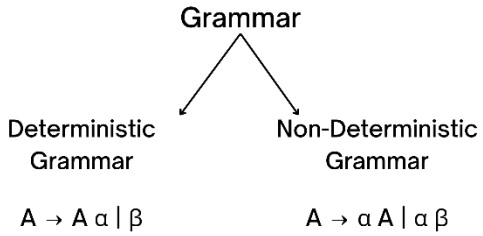
$E' \rightarrow \varepsilon \mid +\, T\, E'$

Left Factoring (LF)

Left factoring is a process by which the grammar with common prefixes is transformed to make it useful for Top down parsers.

In left factoring:

- We make one production for each common prefixes.
- The common prefix may be a terminal or a non-terminal or a combination of both.
- Rest of the derivation is added by new productions.
- The grammar obtained after the process of left factoring is called as Left Factored Grammar.



Example:

1. Do left factoring in the following grammar:

$$S \rightarrow iEtS \mid iEtSeS \mid a$$
$$E \rightarrow b$$

Solution:

The left factored grammar is:

$$S \rightarrow iEtSS' \mid a$$
$$S' \rightarrow eS \mid \varepsilon$$
$$E \rightarrow b$$

2. Do left factoring in the following grammar:

$$A \rightarrow aAB \mid aBc \mid aAc$$

Solution:

$$A \rightarrow aA'$$
$$A' \rightarrow AB \mid Bc \mid Ac$$

Again, this is a grammar with common prefixes.

$$A \rightarrow aA'$$
$$A' \rightarrow AD \mid Bc$$
$$D \rightarrow B \mid c$$

This is a left factored grammar.

Practice Questions

1. Eliminate left recursion from a given grammar.
2. Perform left factoring on an ambiguous grammar.
3. Compare direct and indirect left recursion.
4. Explain the need for left factoring in parsing.

5. Write a grammar with left recursion and factor it.

Viva Questions

1. What is left recursion? Why is it problematic?
2. Differentiate between immediate and indirect left recursion.
3. How do you eliminate left recursion?
4. What is left factoring? Why is it needed?
5. Give examples of left recursion and left factoring.
6. Can LL(1) parsers handle left recursion?
7. Is it possible to remove indirect left recursion in all cases?

7

First(), Follow(), & LL(1)

In compiler design, syntax analysis (parsing) verifies the grammatical structure of the source code. To implement a top-down parser like LL(1), we rely heavily on First() and Follow() sets to construct a predictive parsing table.

FIRST & FOLLOW

FIRST() Set

The First() set of a symbol is the set of terminals that begin the strings derivable from that symbol.

1. If x is a terminal, then FIRST(x) = { 'x' }
2. If x-> Є, is a production rule, then add Є to FIRST(x).
3. If X->Y1 Y2 Y3....Yn is a production,
   a. FIRST(X) = FIRST(Y1)
   b. If FIRST(Y1) contains Є then FIRST(X) = { FIRST(Y1) – Є } U { FIRST(Y2) }
   c. If FIRST (Yi) contains Є for all i = 1 to n, then add Є to FIRST(X).

Example:

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

FIRST() set will be:

$$First(A) = \{a, \varepsilon\}$$
$$First(B) = \{b, \varepsilon\}$$
$$First(S) = \{a, \varepsilon\}$$

FOLLOW() Set

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.

1. Add $ (end of input) to Follow(S), where S is the start symbol.
2. If there is a production $A \rightarrow \alpha B \beta$, then everything in First(β) (except ε) is in Follow(B)
3. If $A \rightarrow \alpha B$ or $A \rightarrow \alpha B \beta$ and First(β) contains ε, then add Follow(A) to Follow(B)

Example:

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

FOLLOW() set will be:

$$\text{Follow(S)} = \{ \$ \}$$
$$\text{Follow(A)} = \text{First(B)} = \{ b \}$$
$$\text{Follow(B)} = \text{Follow(S)} = \{ \$ \}$$

Example:

Production rules are given, Now find sets of FIRST & FOLLOW:

$$E \rightarrow TE'$$
$$E' \rightarrow +T\,E' \mid \varepsilon$$
$$T \rightarrow F\,T'$$
$$T' \rightarrow *F\,T' \mid \varepsilon$$
$$F \rightarrow id \mid (E)$$

Solution:

	First()	Follow()
E	{ (, id }	{ $,) }
E '	{ +, ε }	{ $,) }
T	{ (, id }	{ + , $,) }
T '	{ *, ε }	{ + , $,) }
F	{ *, ε }	{ *, +, $,) }

LL(1) Parsing

An LL(1) parsing table is a table used by a predictive parser. The parser uses it to make decisions using only one lookahead symbol.

Construction Steps

1. For each production $A \rightarrow \alpha$:
2. For each terminal $a \in First(\alpha)$, add $A \rightarrow \alpha$ to M[A,a]
3. If $\varepsilon \in First(\alpha)$, add $A \rightarrow \alpha$ to M[A,b] for all $b \in Follow(A)$

Example:
Construct the LL(1) parsing table using the First-Follow table:

	First()	Follow()
E	{ (, id }	{ $,) }
E'	{ +, ε }	{ $,) }
T	{ (, id }	{ + , $,) }
T '	{ *, ε }	{ + , $,) }
F	{ *, ε }	{ *, +, $,) }

Solution:

Non-terminal	id	*	+	(	)	$
E	E → TE'			E → TE'		
E '			E' → +T E'		E' → ε	E' → ε
T	T -> F T'			T → F T'		
T '		T' → *F T'	T' → ε		T' → ε	T' → ε
F	F → id			F → (E)		

Practice Questions

1. Compute FIRST and FOLLOW sets for a given grammar.
2. Construct the LL(1) parsing table.
3. Analyze if a grammar is LL(1) compatible.
4. Identify conflicts in an LL(1) table.
5. Derive a string using LL(1) parser.

Viva Questions

1. What is the difference between FIRST and FOLLOW?
2. How is the First set used in LL(1) parsing?
3. What does ε (epsilon) signify in grammars?
4. When do you include Follow(A) into Follow(B)?
5. How is an LL(1) parsing table constructed?
6. Why must we remove left recursion for LL(1)?
7. What happens if a cell in the LL(1) table has multiple entries?

8

LR(0) Parser & Canonical Table

LR parsers are a type of bottom-up parser used for syntax analysis in compiler design. They are capable of parsing a large class of context-free grammars and are widely used in programming language compilers due to their efficiency and accuracy.

The term LR(0) stands for:
L: Left-to-right scan of input
R: Rightmost derivation in reverse
0: 0 lookahead symbols used during parsing

LR parsers are among the most powerful and efficient parsers used in compiler design. They work by reading the input from left to right and constructing a rightmost derivation in reverse. Unlike top-down parsers, LR parsers can handle a wider range of grammars, including those with left recursion and ambiguity. This makes them highly suitable for real-world programming languages and robust syntax analysis.

Definition

An LR(0) item is a production with a dot (•) inserted somewhere in the right-hand side (RHS) to indicate how much of the production has been seen.

Example:
For the production:

$$A \rightarrow XYZ$$

We can create the following LR(0) items:

1. $A \rightarrow \bullet XYZ$ (Nothing seen yet)
2. $A \rightarrow X\bullet YZ$ (X has been seen)
3. $A \rightarrow XY\bullet Z$ (X and Y have been seen)
4. $A \rightarrow XYZ\bullet$ (Entire RHS has been seen; ready to reduce)

Closure

The closure of a set of LR(0) items includes those items and adds more based on the following rule:

If $A \rightarrow \alpha \bullet B\ \beta$ is in the item set and $B \rightarrow \gamma$ is a production, then add $B \rightarrow \bullet\gamma$ to the item set.

This process continues until no more new items can be added.

GOTO Function

GOTO(I, X) = The set of items obtained by advancing the dot over symbol X in all items in set I, then computing the closure.

Canonical Collection of LR(0) Items

To build the canonical collection, follow these steps:

1. Start with the augmented grammar, adding a new start symbol $S' \rightarrow S$.
2. Compute closure(I_0), where I_0 = closure($S' \rightarrow \bullet S$)
3. Repeatedly compute GOTO(I, X) for all items I and grammar symbols X
4. Continue until no new item sets are generated

Example:

Given That,

$$S \rightarrow A\ A$$

$$A \rightarrow a\ A \mid b$$

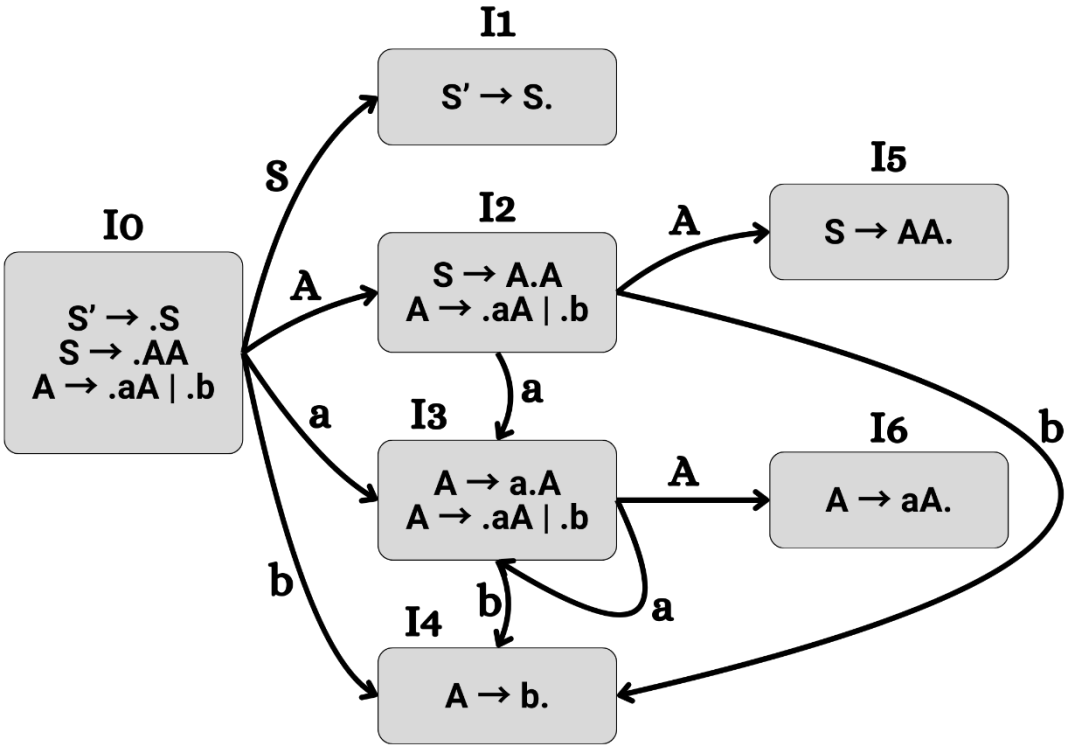
We will denote an expression as:

$$S' \rightarrow .S$$

$$S \rightarrow .A\ A$$

$$A \rightarrow .a\ A \mid .b$$

So, the canonical view:



So, the canonical view:

	Action			Goto	
	a	b	$	A	S
I0	S3	S4		2	1
I1			Accepted		
I2	S3	S4		5	
I3	S3	S4		6	
I4	r3	r3	r3		
I5	r1	r1	r1		
I6	r2	r2	r2		

0	a	3	a	3	b	4	A	6	A	6	A	2	b	4	A	5	S	1

Practice Questions

1. Construct the canonical collection of LR(0) items.
2. Build an LR(0) parsing table for a grammar.
3. Detect shift/reduce conflict in LR(0).
4. Compare LR(0) and LL(1) parsers.
5. Derive a string using LR(0) table.

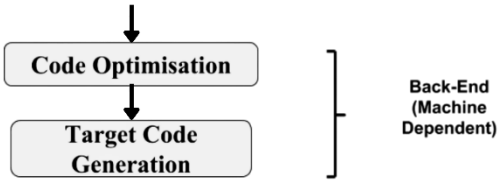
Viva Questions

1. What is LR(0) item?
2. What is the closure of a set of items?
3. How is GOTO different from closure?
4. Why do we augment the grammar?
5. What is a canonical collection?
6. What types of conflicts can occur in LR(0)?
7. What are the limitations of LR(0)?
8. What is the difference between SLR and LR(0)?

9

Intermediate Code Generation

In the analysis-synthesis model of a compiler, the front end of a compiler translates a source program into an independent intermediate code, then the back end of the compiler uses this intermediate code to generate the target code (which can be understood by the machine).



Three address code in Compiler

Three address code is a type of intermediate code which is easy to generate and can be easily converted to machine code. It makes use of at most three addresses and one operator to represent an expression and the value computed at each instruction is stored in temporary variable generated by compiler. The compiler decides the order of operation given by three address code.

Example

Three address code for the following expression: a + b * c – d / (b * c)

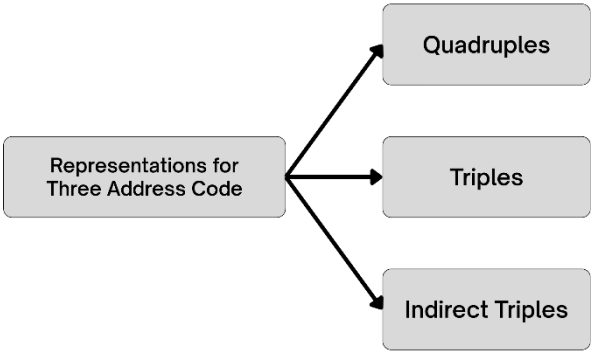
t1 = b * c
t2 = a + t1

t3 = b * c
t4 = d / t3
t5 = t2 – t4

Implementation of Three Address Code

There are 3 representations of three address code:

1. Quadruple
2. Triples
3. Indirect Triples



Quadruple:

For the following expression: a + b * c – d / (b * c)

Three Address Code:

t1 = b * c
t2 = a + t1
t3 = b * c
t4 = d / t3
t5 = t2 – t4

Table: Quadruple table

#	op	arg1	arg2	result
0	*	b	c	t1
1	+	a	t1	t2
2	*	b	c	t3
3	/	d	t3	t4
4	-	t2	t4	t5

Triples:

Table: Triples table

#	op	arg1	arg2
0	*	b	c
1	+	a	(0)
2	*	b	c
3	/	d	(2)
4	-	(1)	(3)

Indirect Triples:

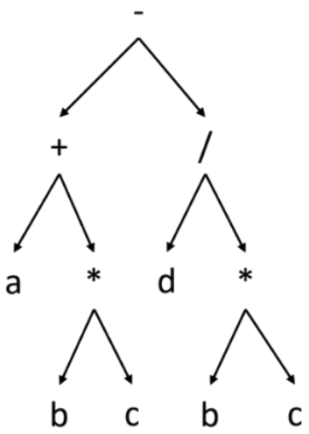
Table: Indirect Triples table

#	Statement
0	21
1	22
2	23
3	24
4	25

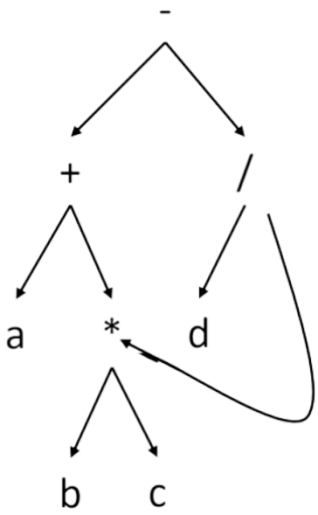
#	op	arg1	arg2
0	*	b	c
1	+	a	(0)
2	*	b	c
3	/	d	(2)
4	-	(1)	(3)

Directed Acyclic Graph (DAG)

For the following expression: a + b * c – d / (b * c)



Syntax directed tree



Final DAG

Practice Questions

1. Generate three-address code for arithmetic expressions.
2. Represent code using quadruples and triples.
3. Explain DAG representation of expressions.
4. Convert infix expressions into TAC.
5. Optimize TAC using temporary variables.

Viva Questions

1. What is Three Address Code?
2. Why is TAC used in compilers?
3. How is TAC different from machine code?
4. What are temporary variables in TAC?
5. How are control flow statements represented in TAC?
6. Can TAC help in optimization?
7. Give an example of TAC for a simple arithmetic statement.

10

Basic Block

In the intermediate representation of programs, a Basic Block is a sequence of consecutive statements in which flow of control enters at the beginning and leaves at the end without any halt or possibility of branching, except at the end. Basic blocks are the foundation for control flow analysis and optimization in compiler design.

Definition

1. A Basic Block (BB) is defined as:
2. A straight-line code sequence.
3. No jumps into or out of the middle of the block.
4. The flow of control enters at the beginning and exits at the end only.

Leader and Basic Block Identification

How to select LEADER

1. The first three address instruction in the intermediate code is a leader.
2. Any instruction that is the target of a conditional or unconditional jump is a leader.
3. Any instruction that immediately follows of a conditional or unconditional jump is a leader.

Algorithm for Finding Basic Blocks

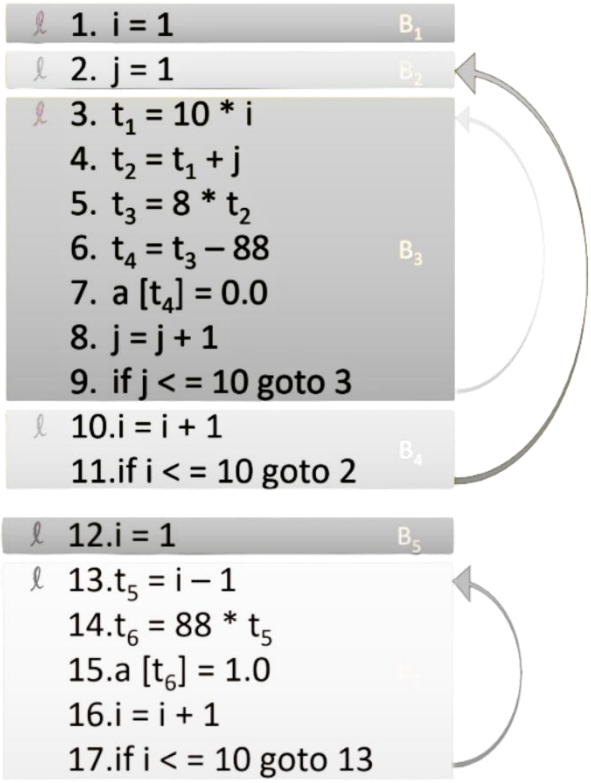
1. Identify all leaders.

2. For each leader, its basic block includes the leader and all statements up to (but not including) the next leader or the end of the program.

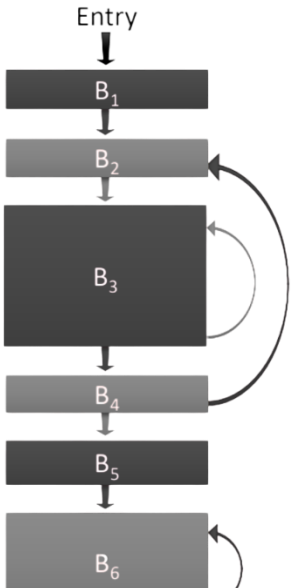
Example:

1. i = 1 2. j = 1 3. t1 = 10 * i 4. t2 = t1 + j 5. t3 = 8 * t2 6. t4 = t3 - 88 7. a[t4] = 0.0 8. j = j + 1 9. if j <= 10 goto 3 10. i = i + 1 11. if i <= 10 goto 2 12. i = 1 13. t5 = i - 1 14. t6 = 88 * t5	15. a[t6] = 1.0 16. i = i + 1 17. if i <= 10 goto 13

Solution:


1. i = 1
B1
2. j = 1
B2
3. t1 = 10 * i
4. t2 = t1 + j
5. t3 = 8 * t2
6. t4 = t3 – 88
7. a [t4] = 0.0
8. j = j + 1
9. if j < = 10 goto 3
B3
10.i = i + 1
11.if i < = 10 goto 2
B4
12.i = 1
B5
13.t5 = i – 1
14.t6 = 88 * t5
15.a [t6] = 1.0
16.i = i + 1
17.if i < = 10 goto 13

Block View:


Entry
B1
B2
B3
B4
B5
B6

Practice Questions

1. Identify basic blocks from a given code segment.
2. Find leaders and construct flow graphs
3. Differentiate between basic block and flow graph.
4. Apply common subexpression elimination in a basic block.
5. Draw DAG for a basic block.

Viva Questions

1. What is a basic block in compiler design?
2. How do you identify leaders?
3. What is the role of basic blocks in optimization?
4. What is a control flow graph?
5. How are conditional statements represented in CFG?
6. Name some optimizations that apply within a basic block.

11

Code Optimization

Code optimization is the process of improving the intermediate or machine code generated by the compiler to make it more efficient without changing its output or functionality. The goal is to reduce the execution time, memory usage, or other computational resources.
Optimization can be performed at various stages of compilation and is crucial in enhancing the performance of compiled programs.

Need for Code Optimization

- To improve execution speed
- To reduce memory consumption
- To generate faster, smaller binaries
- To enhance power efficiency (especially for embedded systems)
- To eliminate redundant operations

Requirements

• Safety
– Preserve the semantics of the program
• Profitability
– Will it help our metric?
• Risk
– How will interact with other optimizations?
– How will it affect other stages of compilation?

Example: Loop Unrolling
• Safety:

– Always safe; getting loop conditions right can be tricky.
• Profitability
– Depends on hardware – usually a win
• Risk
– Increases size of code in loop
– May not fit in the instruction cache

Categories
• Traditional optimizations: Transform the program to reduce work, don't change the level of abstraction
• Enabling transformations: Don't necessarily improve code on their own, Inlining, loop unrolling
• Resource allocation: Map program to specific hardware properties, Register allocation, Instruction scheduling, parallelism, Data streaming, prefetching.

Types of Code Optimization

Machine-Independent Optimization
Performed on the intermediate code, regardless of the target machine.
Examples:
- Constant folding
- Common subexpression elimination
- Loop optimization
- Dead code elimination

Machine-Dependent Optimization
Performed on the machine code and tailored for the architecture of the target CPU.
Examples:
- Instruction scheduling
- Register allocation
- Peephole optimization

Common Code Optimization Techniques

Constant Folding

Evaluates constant expressions at compile time.

Before	After
a = 5 * 2;	a = 10;

Constant Propagation

Replaces variables with their known constant values.

Before	After
int a = 10; int b = a + 5;	int b = 10 + 5;

Dead Code Elimination

Removes code that does not affect the program output.

Before	After
int x = 10; x = 20;	x = 20;

Common Subexpression Elimination

Eliminates expressions that are computed more than once.

Before	After

a = (b + c) + d;
e = (b + c) * f;
	t = b + c;
	a = t + d;
	e = t * f;

Strength Reduction

Replaces expensive operations with cheaper ones.

Before	After

for (int i = 0; i < n; i++)
  a[i] = i * 2;
	int t = 0;
	for (int i = 0; i < n; i++) {
	  a[i] = t;
	  t = t + 2;
	}

Loop Optimization

a) Loop Invariant Code Motion

Moves calculations that do not change inside loops to outside.

Before	After
for (int i = 0; i < n; i++) { x = a * b;	x = a * b; for (int i = 0; i < n; i++) {

arr[i] = x + i; }	arr[i] = x + i; }

b) Loop Unrolling
Reduces the overhead of loop control by duplicating the loop body.

c) Loop Fusion & Fission
Fusion: Combines adjacent loops
Fission: Splits a loop into multiple

Peephole Optimization

A local optimization technique that examines a small set of instructions (window) and replaces them with more efficient code.

Before	After
MOV A, B MOV B, A	(No Operation)

Optimization Trade-Offers
May increase compilation time
Can increase code size (e.g., unrolling)
Requires correctness preservation
Optimizations must be balanced with resource constraints

Example: Optimized Three Address Code

Before	t1 = a + b
t1 = a + b t2 = a + b t3 = t1 * t2	t3 = t1 * t1

Tools for Code Optimization
- LLVM (Low-Level Virtual Machine)
- GCC (GNU Compiler Collection)
- Clang
- Custom optimization passes in academic compilers

Practice Questions

1. Apply constant folding and propagation.
2. Identify dead code and eliminate it.
3. Optimize loop structures in code.
4. Demonstrate strength reduction in a program.
5. Apply peephole optimization techniques.

Viva Questions

1. What is code optimization?
2. What is the difference between machine-independent and machine-dependent optimization?
3. What is loop invariant code motion?
4. What is the purpose of common subexpression elimination?
5. Define peephole optimization with an example.
6. Why is code optimization important?
7. What is dead code elimination?

Compiler Design Lab

CODE PART

Basic C programming

Basic Input/Output Operations

1. Write a C program to read a full sentence using scanf() and print it.

Solution:

```c
#include <stdio.h>
int main() {
char sentence[100];
printf("Enter a sentence: ");
scanf("%[^\n]", sentence);   // Reads the full line until newline
printf("You entered: %s\n", sentence);
return 0;
}
```

Output:

```
Enter a sentence: Hi, I'm Sajjad.
You entered: Hi, I'm Sajjad.
```

2. Write a C program that reads a number and checks whether it is even or odd.

Solution:

```c
#include <stdio.h>
int main() {
    int num;
    printf("Enter a number: ");
    scanf("%d", &num);
    if (num % 2 == 0) {
        printf("%d is Even.\n", num);
    } else {
        printf("%d is Odd.\n", num);
     }
     return 0;
}
```

Output:

```
Enter a number: 2025
2025 is Odd.
```

Basic Array Operations

1. Write a C program to reverse an array.

Solution:

```c
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 3, 4, 5}, n = 5;
    printf("Original array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);
    // Reversing the array
    for (int i = 0; i < n / 2; i++) {
        int temp = arr[i];
        arr[i] = arr[n - 1 - i];
        arr[n - 1 - i] = temp;
     }
    printf("\nReversed array: ");
    for (int i = 0; i < n; i++)
        printf("%d ", arr[i]);

return 0;
}
```

Output:

```
Original array: 1 2 3 4 5
Reversed array: 5 4 3 2 1
```

2. Write a C program to remove duplicate elements from an array.

Solution:

```c
#include <stdio.h>

int main() {
    int arr[] = {1, 2, 2, 3, 4, 4, 5}, n = 7;
    int temp[100], index = 0;

    for (int i = 0; i < n; i++) {
        int isDuplicate = 0;
        for (int j = 0; j < index; j++) {
            if (arr[i] == temp[j]) {
                isDuplicate = 1;
                break;
            }
        }
        if (!isDuplicate)
            temp[index++] = arr[i];
    }
    printf("Array after removing duplicates: ");
    for (int i = 0; i < index; i++)
        printf("%d ", temp[i]);

    return 0;
}
```

Output:

```
Array after removing duplicates: 1 2 3 4 5
```

3. Write a C program to sort an array in ascending and descending order.

Solution:

```c
#include <stdio.h>

int main() {
    int arr[] = {5, 2, 8, 1, 3};
    int n = sizeof(arr) / sizeof(arr[0]);
```

```
    int i, j, temp;
    // Ascending sort
    for (i = 0; i < n - 1; i++)
        for (j = 0; j < n - i - 1; j++)
            if (arr[j] > arr[j + 1]) {
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }

     printf("Ascending: ");
     for (i = 0; i < n; i++)
         printf("%d ", arr[i]);

     // Descending sort (on same array)
     for (i = 0; i < n - 1; i++)
         for (j = 0; j < n - i - 1; j++)
             if (arr[j] < arr[j + 1]) {
                 temp = arr[j];
                 arr[j] = arr[j + 1];
                 arr[j + 1] = temp;
             }

    printf("\nDescending: ");
     for (i = 0; i < n; i++)
         printf("%d ", arr[i]);

     return 0;
}
```

Output:

```
Ascending: 1 2 3 5 8
Descending: 8 5 3 2 1
```

Basic String Operations

1. Write C a program to find the length of a string without using strlen().

Solution:

```
#include <stdio.h>

int main() {
    char str[100];
    int length = 0;

    printf("Enter a string: ");
    gets(str);  // Or use fgets for safer input

    while (str[length] != '\0')
           length++;
   printf("Length of the string: %d\n", length);
   return 0;
}
```

Output:

```
Enter a string: Daffodil International University
Length of the string: 33
```

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

Solution:

```
#include <stdio.h>

int main() {
    char str[100];
    int count = 0;
    printf("Enter a string: ");
    gets(str);  // Or use fgets
    for (int i = 0; str[i] != '\0'; i++) {
        if (str[i] == ' ')
            count++;
```

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

Output:

```
Enter a string: Counting  White Sp ace .
Number of white spaces: 5
```

3. C program that will remove white space from a string.

Solution:

```
#include <stdio.h>

int main() {
    char str[100], result[100];
    int j = 0;
    printf("Enter a string: ");
    gets(str);

    for (int i = 0; str[i] != '\0'; i++) {
        if (str[i] != ' ') {
            result[j++] = str[i];
        }
    }
    result[j] = '\0';
    printf("String without spaces: %s\n", result);

    return 0;
}
```

Output:

```
Enter a string: Re moving white Sp  ace.
String without spaces: RemovingwhiteSpace.
```

File Handling Basics

1. C program to read a file and display its content on the screen.

Solution:

```
#include <stdio.h>
int main() {

    FILE *file;
    char ch;
    file = fopen("example.txt", "r");

    if (file == NULL) {
        printf("Error: Could not open the file.\n");
        return 1;
     }
     printf("File content:\n");
     while ((ch = fgetc(file)) != EOF) {
         putchar(ch);
     }
     fclose(file);
     return 0;
}
```

Output:

```
File content:
Hello World!
This is a test file.
```

Practice Problems

1. Write a program to copy one string to another without using strcpy().
2. Write a program to count the number of digits, alphabets, and special characters in a string.
3. Write a program to count occurrences of a particular word in a given text.
4. Write a program to merge two sorted arrays into a single sorted array.

Viva Questions

1. What is the difference between scanf() and gets() in C?
2. How does strlen() work internally?
3. What is the use of fopen(), fclose(), fprintf(), and fscanf()?

Lexemes, Tokens & Patterns

Basic Tokenization

1. Write a C program to count the number of words in a given string.

Solution:

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

int main() {
    char str[1000];
    int i, words = 0, inWord = 0;
    printf("Enter a sentence: ");

     fgets(str, sizeof(str), stdin);

     for (i = 0; str[i] != '\0'; i++)
    {
         if (isspace(str[i])) {
             inWord = 0;
         } else if (!inWord) {
             inWord = 1;
             words++;
         }
     }

     printf("Total words: %d\n", words);
     return 0;
}
```

Output:

```
Enter a sentence: Compiler Desing Lab
Total words: 3
```

2. Write a C program to count and print the number of keywords in a given C source code.

Solution:

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

const char *keywords[] = {"int", "float", "if", "else", "for", "while", "return", "void", "char", "double", "long", "short", "switch", "case", "break", "continue", "default", "do", "signed", "unsigned", "struct", "union", "enum", "typedef", "const", "volatile", "static", "extern", "goto", "sizeof", "auto", "register"
};

int isKeyword(const char *word) {
  for(int i=0; i<sizeof(keywords)/sizeof(keywords[0]);i++){
        if (strcmp(word, keywords[i]) == 0)
            return i;  // return index of the keyword
    }
    return -1;
}

int main() {
    char code[1000], word[50];
    int i = 0, j = 0;
    int keywordFlags[100] = {0};
    printf("Enter C code (end with $):\n");
    while ((code[i] = getchar()) != '$') i++;
    code[i] = '\0';
    int total = 0;
    for (i = 0; code[i] != '\0'; i++) {
        if (isalnum(code[i]) || code[i] == '_') {
            word[j++] = code[i];
        } else {

```

```
            if (j > 0) {
                word[j] = '\0';
                int index = isKeyword(word);
                 if (index != -1 && keywordFlags[index] ==
  0) {
                    printf("Keyword found: %s\n", word);
                    keywordFlags[index] = 1;
                    total++;
                }
                j = 0;
            }
        }
    }
    printf("Total unique keywords: %d\n", total);
    return 0;
}
```

Output:

```
Enter C code (end with $):
int main() {
    int a = 5;
    float b = 3.2;
    return 0;
}$
Keyword found: int
Keyword found: float
Keyword found: return
Total unique keywords: 3
```

3. Write a C program that can detect and count all numeric constants in a string.

Solution:

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

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

```

```
    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

     while (str[i]) {
         if (isdigit(str[i])) {
             count++;
             while (isdigit(str[i]) || str[i] == '.') i++;
         } else {
             i++;
         }
     }
     printf("Total numeric constants: %d\n", count);
     return 0;
}
```

Output:

```
Enter a string: It's 2025, of L4T1(spring semester) beginning
of the year.
Total numeric constants: 3
```

Comment Handling

1. Write a program to count the number of comment lines in a C source code.

Solution:

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

int main() {
    char code[1000];
    int i = 0, count = 0;
    printf("Enter C code (end with $):\n");

    while ((code[i] = getchar()) != '$') i++;
     code[i] = '\0';
```

```
    for (i = 0; code[i] != '\0'; i++) {
        if (code[i] == '/' && code[i + 1] == '/') {
            count++;
            while (code[i] != '\n' && code[i] != '\0') i++;
        }
        else if (code[i] == '/' && code[i + 1] == '*') {
            count++;
            i += 2;

            while (!(code[i] == '*' && code[i + 1] == '/')
&& code[i] != '\0') {
                if (code[i] == '\n') count++;
                i++;
            }
        }
    }

    printf("Total comment lines: %d\n", count);

    return 0;
}
```

Output:

```
Enter C code (end with $):
// This is a single-line comment
int a = 10; /* This is
a multi-line
comment */
printf("Hello"); // another comment
$
Total comment lines: 5
```

2. Write a program to eliminate single-line (//) comments from a C source code.

Solution:

```
#include <stdio.h>
int main() {
    char code[1000];
    int i = 0;
```

```
    printf("Enter C code (end with $):\n");
    while ((code[i] = getchar()) != '$') i++;
    code[i] = '\0';

    printf("\nCode without single-line comments:\n");
     for (i = 0; code[i] != '\0'; i++) {
         if (code[i] == '/' && code[i + 1] == '/') {
             while (code[i] != '\n' && code[i] != '\0') i++;
         }
     if (code[i] != '\0')
             i.  putchar(code[i]);
     }
     return 0;
}
```

Output:

```
Enter C code (end with $):
// This is a single-line comment
int a = 10;
printf("Hello"); // another comment
$

Code without single-line comments:

int a = 10;
printf("Hello");
```

3. Write a program to eliminate multi-line (/* ... */) comments from a C source code.

Solution:

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

int main() {
char code[1000];
int i = 0;

printf("Enter C code (end with $):\n");
```

```
while ((code[i] = getchar()) != '$') i++;
code[i] = '\0';
printf("\nCode without multi-line comments:\n");

    for (i = 0; code[i] != '\0'; i++) {
        if (code[i] == '/' && code[i + 1] == '*') {
            i += 2;
             while (!(code[i] == '*' && code[i + 1] == '/') &&
code[i] != '\0') i++;
            i += 1; // skip closing */
        } else {
            putchar(code[i]);
        }
    }
    return 0;
}
```

Output:

```
Enter C code (end with $):
int a = 10; /* This is
a multi-line
comment */
printf("Hello");
$

Code without multi-line comments:
int a = 10;
printf("Hello");
```

Token Classification

1. Write a program that identifies and prints all punctuations from a given C code snippet.

Solution:

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

int isPunctuation(char ch) {
    char punctuations[] = "();:,{}[]#\"'\\";
    for (int i = 0; i < strlen(punctuations); i++) {
        if (ch == punctuations[i])
            return 1;
    }
     return 0;
}

int main() {
    char code[1000];
    int i = 0;

    printf("Enter C code (end with $):\n");
    while ((code[i] = getchar()) != '$') i++;
    code[i] = '\0';

    printf("\nPunctuations found:\n");
    for (i = 0; code[i] != '\0'; i++) {
        if (isPunctuation(code[i])) {
            printf("%c ", code[i]);
        }
    }
    return 0;
}
```

Output:

```
Enter C code (end with $):
int a = 42;
float pi = 3.14;
char ch = ';';
$

Punctuations found:
; ; ' ; ' ;
```

2. Write a program that prints all integer and floating-point constants from a string.

Solution:

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

int main() {
    char input[1000];
    int i = 0;
    printf("Enter the string (end with $):\n");
    while ((input[i] = getchar()) != '$') i++;
    input[i] = '\0';

     printf("\nConstants found:\n");
     i = 0;
     while (input[i] != '\0') {
         if (isdigit(input[i])) {
             int hasDot = 0;
             while (isdigit(input[i]) || input[i] == '.') {
                 if (input[i] == '.')
                     hasDot = 1;
                 putchar(input[i]);
                 i++;
             }
             if (hasDot)
                 printf(" (float)\n");
             else
                 printf(" (int)\n");
```

```
        } else {
            i++;
        }
    }
}
```

Output:

```
Enter the string (end with $):
int a = 42;
float pi = 3.14;
char ch = ';';
$

Constants found:
42 (int)
3.14 (float)
```

Practice Problems

1. Write a C program to split a given sentence into words using custom logic (not strtok()).
2. Write a C program that removes all spaces, tabs, and newlines from a file.
3. Write a program that accepts a line of C code and counts different types of brackets.

Viva Questions

1. What is a token in compiler design?
2. What is the role of a lexical analyzer in a compiler?
3. What are keywords, identifiers, operators, and literals?
4. What is the difference between a lexical error and a syntax error?
5. What are lexemes? How are they different from tokens?

Analysis of Regular Expressions

Problem Statements

1. Write a C program that will check inputs accepted / rejected by the RE $(ab)^{2+}$.

Solution:

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

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

     int len = strlen(str);
     bool isValid = true;

     if (len < 4 || len % 2 != 0) {
         isValid = false;
     } else {
         for (int i = 0; i < len; i += 2) {
             if (str[i] != 'a' || str[i + 1] != 'b') {
                 isValid = false;
                 break;
             }
         }

         if ((len / 2) < 2) {
             isValid = false;
         }
     }
```

```
    if (isValid) {
        printf("Accepted\n");
    } else {
        printf("Rejected\n");
    }
    return 0;
}
```

Output:

```
Enter the string: abababab
Accepted
```

2. Write a C program to simulate the pattern ab* + (ba)*.

Solution:

```
#include <stdio.h>
#include <string.h>
#include <stdbool.h>
bool is_ab_star(const char *str) {
    if (str[0] != 'a') return false;
    for (int i = 1; str[i]; i++) {
        if (str[i] != 'b') return false;
    }
    return true;
}
bool is_ba_star(const char *str) {
    int len = strlen(str);
    if (len % 2 != 0) return false;
    for (int i = 0; i < len; i += 2) {
        if (str[i] != 'b' || str[i + 1] != 'a') return false;
    }
    return true;
}
int main() {
    char str[100];
    printf("Enter the string: ");
    scanf("%s", str);
```

```
    if (is_ab_star(str) || is_ba_star(str))
        printf("Accepted\n");
    else
        printf("Rejected\n");

    return 0;
}
```

Output:

```
Enter the string: a
Accepted
```

Practice Problems

1. Simulate REs like (a+b)*abb.
2. Match valid email pattern using RE logic.
3. Match variable naming conventions using RE.

Viva Questions

1. What is a regular expression?
2. How does RE differ from CFG?
3. What are the operators used in RE?
4. How can RE be used in lexical analysis?

CFG Analysis — First(), Follow()

Problem Statements

1. Write a C program to compute the FIRST set of a grammar with productions like:

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

Solution:

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

#define MAX 10

char productions[][10] = {
    "E->TE",
    "E->#",
     "T->FT",
     "T->#",
     "F->(E)",
     "F->id"
};

int n = 6; // number of productions
char firstSet[10];
int m = 0;

```

```
void findFirst(char symbol);

void addToFirst(char c) {
    for (int i = 0; i < m; i++) {
        if (firstSet[i] == c)
            return;
    }
    firstSet[m++] = c;
}

void findFirst(char symbol) {
    for (int i = 0; i < n; i++) {
        if (productions[i][0] == symbol) {
            if (!isupper(productions[i][3])) {
                addToFirst(productions[i][3]);
            } else {
                findFirst(productions[i][3]);
            }
        }
    }
}

int main() {
    char symbol = 'E'; // fixed non-terminal to find FIRST(E)
    findFirst(symbol);

    printf("FIRST(%c) = { ", symbol);
    for (int i = 0; i < m; i++) {
        printf("%c ", firstSet[i]);
    }
    printf("}\n");

    return 0;
}
```

Output:

```
FIRST(E) = { ( i # }
```

2. Write a C program to compute the FOLLOW set for the same grammar:

Solution:

```c
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#define MAX 10
char productions[][10] = {
    "E->TE",
    "E->#",
    "T->FT",
    "T->#",
     "F->(E)",
     "F->id"
};

int n = 6;
char followSet[10];
int m = 0;

void addToFollow(char c) {
    for (int i = 0; i < m; i++) {
        if (followSet[i] == c)
            return;
    }
    followSet[m++] = c;
}

void findFollow(char symbol) {
    if (symbol == 'E') {
        addToFollow('$'); // start symbol gets '$' in FOLLOW
    }
```

```
    for (int i = 0; i < n; i++) {
        char* prod = productions[i];
        for (int j = 3; prod[j] != '\0'; j++) {
            if (prod[j] == symbol) {
                if (prod[j + 1] != '\0') {
                    char next = prod[j + 1];
                    if (!isupper(next)) {
                        addToFollow(next);
                    } else {
                        //  Add  FIRST(next)  (simplified:
  assume next leads to id or epsilon)
                        if (next == 'E' || next == 'T') {
                            addToFollow('(');
                            addToFollow('i');
                        }
                        if (next == 'F') {
                            addToFollow('(');
                            addToFollow('i');
                        }
                    }
                } else {
                    if (prod[0] != symbol) {
                        findFollow(prod[0]);
                    }
                }
            }
        }
    }
}

int main() {
    char symbol = 'E'; // fixed non-terminal
    findFollow(symbol);

    printf("FOLLOW(%c) = { ", symbol);
```

```
int main() {
    char symbol = 'E'; // fixed non-terminal
    findFollow(symbol);

    printf("FOLLOW(%c) = { ", symbol);
    for (int i = 0; i < m; i++) {
        printf("%c ", followSet[i]);
    }
    printf("}\n");

    return 0;
}
```

Output:

```
FOLLOW(E) = { $ ) }
```

Practice Problems

1. Write a program to compute FIRST sets of a grammar.
2. Write a program to compute FOLLOW sets of a grammar.
3. Generate LL(1) table from FIRST and FOLLOW.

Viva Questions

1. What is a CFG?
2. How is CFG different from a regular grammar?
3. What are terminal and non-terminal symbols?
4. What is derivation in CFG?
5. What is ambiguity in a grammar?

Left Recursion and Left Factoring

Problem Statements

1. Write a C program to eliminate immediate left recursion for a fixed grammar.

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

Solution:

```c
#include <stdio.h>

int main() {
    // Fixed grammar: A → A a | b
    printf("Original Grammar:\n");
    printf("A -> A a | b\n");

    // Applying immediate left recursion removal:
    // A → b A'
     // A' → a A' | ep    [ep = epsilon(ε), the empty string]

     printf("\nGrammar after eliminating Left Recursion:\n");
     printf("A  -> b A'\n");
     printf("A' -> a A' | ep\n");

     return 0;
}
```

Output:

```
Original Grammar:
A -> A a | b

Grammar after eliminating Left Recursion:
A  -> b A'
A' -> a A' | ep
```

2. Write a C program to perform left factoring for a fixed grammar.

$$S \rightarrow i\,e\,t\,S \mid i\,e\,S \mid a$$

Solution:

```c
#include <stdio.h>

int main() {
    // Original grammar
    printf("Original Grammar:\n");
    printf("S -> i e t S | i e S | a\n");

    // After left factoring:
    // S  → i e S' | a
     // S' → t S | ep [ep = epsilon(ε)]

     printf("\nGrammar after Left Factoring:\n");
     printf("S  -> i e S' | a\n");
     printf("S' -> t S | ep\n");

     return 0;
}
```

Output:

```
Original Grammar:
S -> i e t S | i e S | a

Grammar after Left Factoring:
S  -> i e S' | a
S' -> t S | ep
```

Practice Problems

1. Write a program to eliminate left recursion from a grammar.
2. Apply left factoring to a grammar and display the result.
3. Convert an ambiguous grammar using left recursion elimination.
4. Check whether a grammar has left recursion.
5. Transform a CFG into an LL(1) friendly grammar.

Viva Questions

1. What is direct left recursion?
2. What is indirect left recursion?
3. When is left factoring required?
4. Why is left recursion not suitable for recursive descent parsers?
5. What is the output of left factoring?

C Ɔ _

Project Ideas

These projects help students explore various components of compilers in a practical way.

Web-Based Project Ideas (Compiler-Themed)

These projects blend Compiler Design with Web Technologies like HTML/CSS/JS or backend languages.

1. Online Lexical Analyzer

- Web interface to input code and display tokens.
- Highlight syntax elements (keywords, strings, comments).
- Can be made using JavaScript.

2. Interactive Parser Visualizer

- Enter a grammar and input string.
- Show steps of derivation (LMD, RMD), parse tree.
- Web technologies: JavaScript + D3.js for tree visualization.

3. Web-Based Regular Expression Simulator

- Accept RE input and generate corresponding NFA/DFA.
- Visualize transition diagrams and simulate string acceptance.

4. TinyLang Online IDE

Create a small interpreted language (TinyLang) and run it on a browser.
Show errors (syntax, logical) and intermediate steps.

5. Online TAC Generator

- User inputs a high-level expression.
- Output shows three address code step by step.
- Useful for learning intermediate code generation.

6. Compiler Teaching Portal

- An educational site with interactive lessons on compiler phases.
- Includes quizzes, simulations, and mini-games.

My Project Idea:

P Compiler Forge Toolkit: Web-Based Solution for Simplifying Compiler Theory Concepts

Proposed Architecture

In this chapter, we describe the architectural blueprint of the Compiler Forge Toolkit. It includes the requirement analysis, design considerations, user interface design, and a structured development plan that guided the implementation of the project.

2.1 Requirement Analysis & Design Specification

2.1.1 Overview

Before beginning development, it was essential to identify the system's functional needs and the most suitable design patterns to ensure a smooth user experience. The goal was to develop a lightweight, client-side application capable of processing grammar rules and rendering results in a clean and understandable manner.

2.1.2 System Design

The system design of the Compiler Forge Toolkit is based on a modular, client-side architecture. The entire application is built to run in the user's browser without the need for any backend services. This design ensures fast interaction, real-time results, and accessibility across devices. The architecture can be best understood through its **three key layers**:

a. User Interface Layer

This is the **front-facing layer** where all user interaction takes place. It collects grammar input, allows operation selection, and displays the transformed grammar or calculated sets. The UI is designed to be intuitive and responsive.

Components:

- **Dropdown Selector**: Lets users select the desired operation – Left Recursion Elimination, Left Factoring, or First/Follow Set.
- **Text Input Field**: Allows multiline grammar input in a readable format.
- **Generate Button**: Triggers the processing logic.
- **Output Display Section**: Dynamically renders the results after processing.

This layer ensures a smooth interaction flow, minimizing user confusion through clarity and simplicity.

b. Processing Layer

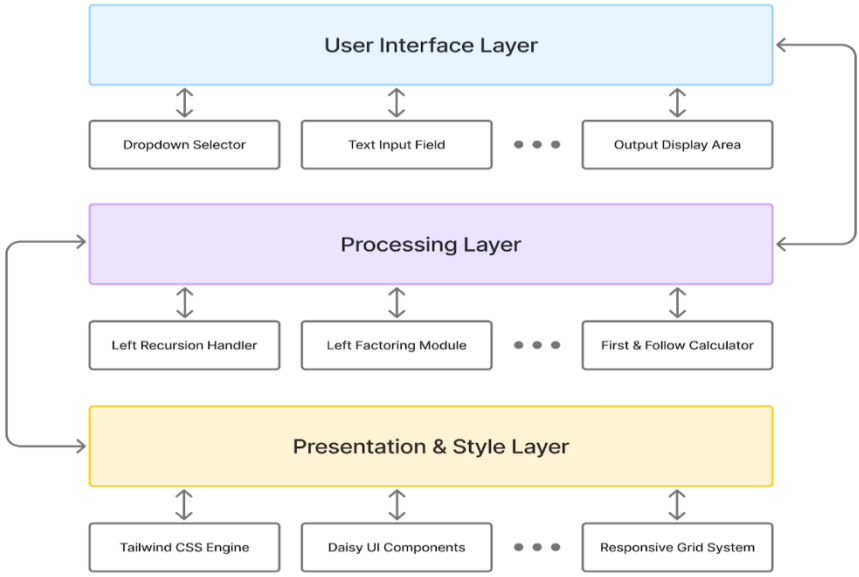
This is the core logic layer where all grammar transformations and set calculations are executed. It is entirely implemented in JavaScript, which makes it suitable for frontend-only execution.

Modules:

- **Left Recursion Handler**: Detects and transforms left-recursive productions into right-recursive forms.
- **Left Factoring Module**: Analyzes and refactors common prefixes to make the grammar suitable for predictive parsing.
- **First & Follow Calculator**: Computes the FIRST and FOLLOW sets required for LL(1) parsing.\
- **Input Validator**: Validates grammar syntax before processing to ensure results are meaningful.

This layer ensures accuracy and efficiency, mimicking classroom procedures with programmatic precision.

c. Presentation & Styling Layer



This layer controls the visual appearance of the web application. The emphasis is on clean typography, readable spacing, and responsive design across various screen sizes.

Technologies Used:

- **Tailwind CSS**: A utility-first CSS framework used to create custom designs without writing conventional CSS.
- **Daisy UI**: Built on top of Tailwind, it provides ready-to-use styled components for rapid UI development.
- **Responsive Grid System**: Ensures adaptability to desktops, tablets, and mobile phones without layout breakage.

This layer brings together design consistency, accessibility, and modern UI aesthetics.

Figure 1: System Architecture of Compiler Forge Toolkit

This architecture enables fast, responsive behavior and allows the tool to function offline or in restricted environments without backend dependency.

2.1.3 User Interface Design

The User Interface (UI) of the Compiler Forge Toolkit has been designed with a clean, user-friendly, and functional layout that supports real-time grammar analysis tasks in an educational setting. The interface emphasizes clarity, accessibility, and responsiveness for diverse users including students, instructors, and researchers.

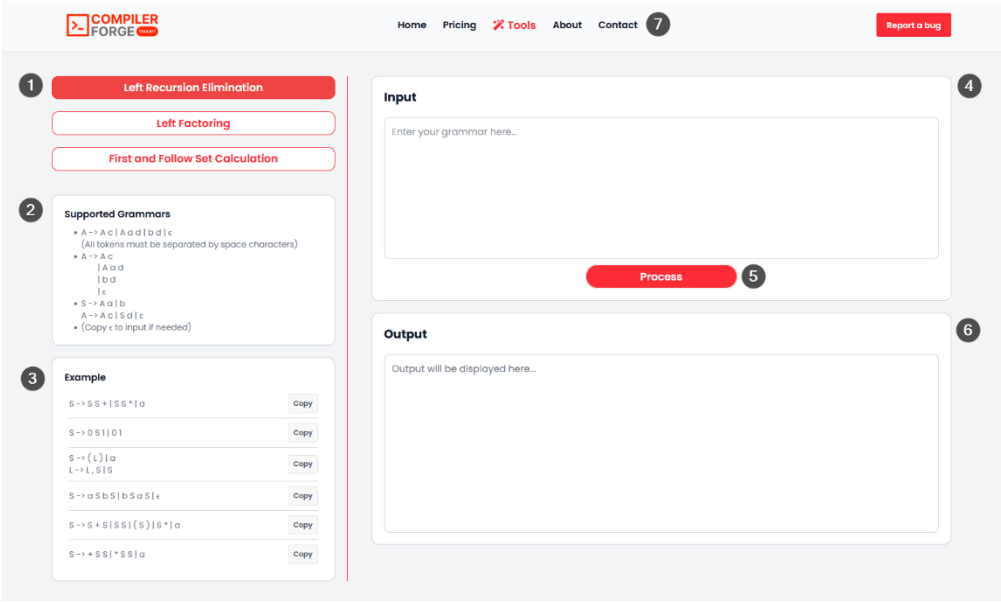
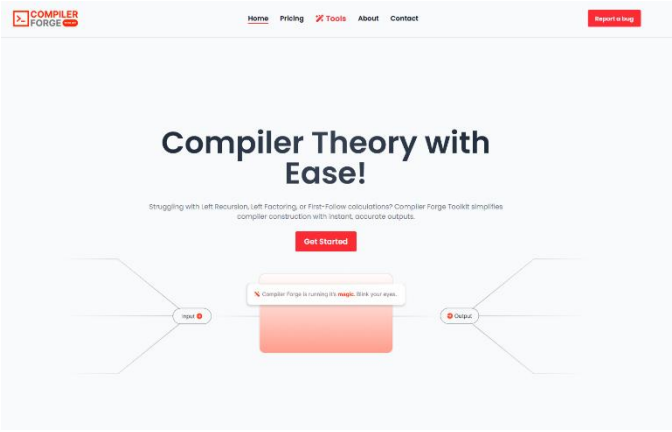


Figure 2: User Interface Of Compiler Forge Toolkit

Key Interface Components:

1. **Tool Selector Buttons (Top Left Section)**
   - The UI offers **three prominent buttons** to switch between available functionalities:
     - **Left Recursion Elimination**
     - **Left Factoring**
     - **First and Follow Set Calculation**
   - These buttons are color-coded for visibility and organized in a vertical stack for intuitive access.
2. **Supported Grammars Panel**
   - A dedicated panel provides examples of grammar formatting and rules, with usage hints such as:
     - All tokens must be separated by space characters.
     - Use of special symbols like ε (epsilon) for empty productions.
   - This aids users in input preparation and prevents formatting errors.
3. **Example Section with Quick Copy**
   - This section lists **predefined grammar examples** relevant to each tool.
   - Each example has an adjacent **"Copy" button** allowing users to quickly use and test grammars without manual input.
4. **Input Area (Top Right Section)**
   - A large text box is provided where users can enter their custom grammar.
   - The placeholder text "Enter your grammar here..." gives clear direction.
5. **Process Button**
   - Centrally placed in the input/output section, this button triggers the backend JavaScript logic to compute and transform the grammar.
   - The red color and rounded styling make it prominent and easily accessible.
6. **Output Display Area**
   - This section dynamically shows the result of the selected operation.
   - Output is presented in a read-only format to ensure consistency and avoid accidental edits.
7. **Navigation and Utility**
   - The top navigation bar includes links to **Home**, **Pricing**, **Tools**, **About**, and **Contact** pages.
   - A **"Report a bug"** button is placed in the top-right corner, facilitating user feedback and iterative improvement.



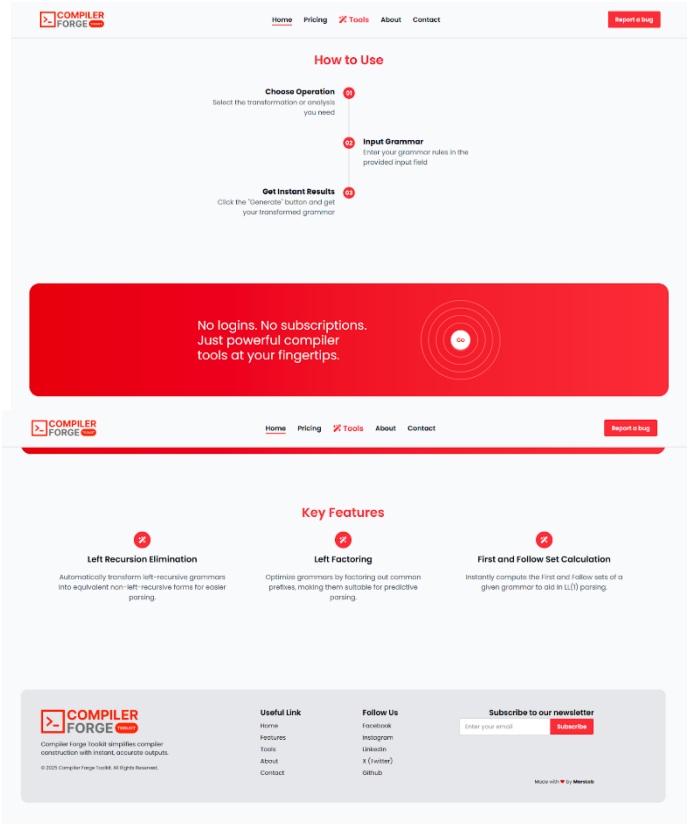


Figure 3: User Interface Of the Home Screen

Key Interface Components:

- **Responsiveness**: Built with Tailwind CSS and Daisy UI, the layout adjusts seamlessly to various screen sizes—from desktops to mobile devices.
- **Clarity**: Minimalist aesthetic with ample white space, soft shadows, and readable font sizes enhances usability.
- **Usability**: Every design choice, from button colors to layout alignment, contributes to reducing user confusion and improving efficiency.

The UI of the Compiler Forge Toolkit serves as the bridge between user intent and algorithmic processing. It not only supports seamless interactions but also educates the user on grammar formatting and processing workflows. The thoughtful arrangement of components ensures that even novice users can interact with compiler design tools confidently.

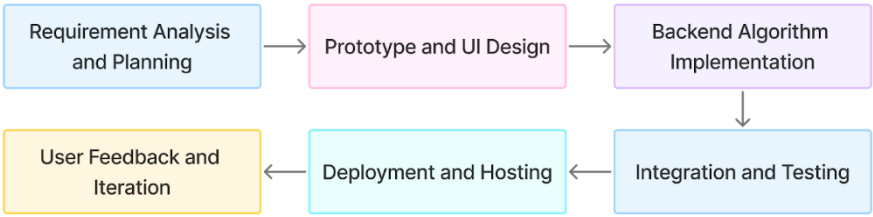
2.2.3 Development Phase

The development process of Compiler Forge Toolkit followed an **iterative and modular approach**, divided into the following key stages:

1. **Requirement Analysis and Planning**
   - Gathered functional and non-functional requirements
   - Identified target users and core use-cases

2. **Prototype and UI Design**
   - Created wireframes and mockups using Figma

Designed user interface with Tailwind CSS and Daisy UI components



3. **Backend Algorithm Implementation**
   - JavaScript-based implementations for grammar processing
   - Focused on accuracy and efficiency of parsing-related computations

4. **Integration and Testing**
   - Connected frontend and backend with event-driven interactions
   - Performed unit testing on grammar rules and edge cases

5. **Deployment and Hosting**
   - Deployed the final application on **Netlify**
   - Ensured public accessibility with a secure and shareable link: https://compiler-forge.netlify.app

6. **User Feedback and Iteration**
   - Incorporated suggestions from peers and instructors
   - Addressed minor bugs and refined UI usability

In summary, the overall project plan for Compiler Forge Toolkit emphasized structured, goal-oriented development aligned with academic utility. With a clear focus on educational value, usability, and modular design, the project ensures continued relevance and usability for students learning compiler design concepts.

Compiler Forge Live Site Link : https://compiler-forge.netlify.app
Compiler Forge GitHub Repository : https://github.com/garodiaa/compiler-forge-toolkit

Compiler Forge Project Directory Structure :

```
COMPILER-PROJECT
├ assets
|  ├ fav_ico.png
|  ├ hero 1.png
|  └ logo_header.png
├ scripts
|  ├ basic.js
|  ├ ff.js
|  ├ index.js
|  ├ lf.js
|  ├ lr.js
|  └ tools.js
├ about.html
├ contact.html
├ index.html
├ pricing.html
├ README.md
├ tailwind.config.js
└ tools.html
```

index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Compiler Forge Toolkit</title>
  <!-- favicon -->
  <link rel="icon" href="assets/fav_ico.png" type="image/x-icon">
  <!-- daisy and tailwind -->
  <link     href="https://cdn.jsdelivr.net/npm/daisyui@5"     rel="stylesheet"
type="text/css" />
  <script src="https://cdn.jsdelivr.net/npm/@tailwindcss/browser@4"></script>
  <!-- google font -->
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <link
href="https://fonts.googleapis.com/css2?family=Poppins:ital,wght@0,100;0,200;
0,300;0,400;0,500;0,600;0,700;0,800;0,900;1,100;1,200;1,300;1,400;1,500;1,600
;1,700;1,800;1,900&display=swap" rel="stylesheet">
  <!-- font awesome -->
  <script                       src="https://kit.fontawesome.com/f6fc430968.js"
crossorigin="anonymous"></script>
</head>
<body class="font-[Poppins] text-gray-900">
       <!-- header -->
       <header class="bg-gray-50 border-b-2 border-gray-200 sticky top-0 z-50
py-5 h-5rem">
       <nav class="px-5 container m-auto flex justify-between items-center">
              <div>
              <a       href="index.html"><img       class="h-8       sm:h-12"
src="assets/logo_header.png" alt=""></a>
              </div>
              <!-- mid -->
              <div>
             <ul class="flex items-center gap-7 font-semibold cursor-pointer">
              <li class="hidden md:block"><a href="index.html" class="border-
b-2 border-red-500 hover:text-red-600">Home</a></li>
              <li     class="hidden     md:block"><a     href="pricing.html"
class="hover:text-red-600">Pricing</a></li>
              <li id="btn-tools" class="py-1 text-lg text-red-500 border-red-
500 transition-all duration-300 hover:scale-110"><i class="fa-solid fa-wand-
magic-sparkles"></i> Tools</li>
              <li      class="hidden      md:block"><a      href="about.html"
class="hover:text-red-600">About</a></li>
              <li class="hidden md:block"><a href="contact.html" hover:text-
red-600">Contact</a></li>
              </ul>
              </div>
              <div>
              <button
              class="btn btn-sm sm:btn-md py-1 bg-red-500 text-white border-
red-500 hover:bg-transparent hover:text-red-500">Report a bug</button>
              </div>
       </nav>
       </header>
```

```
        <!-- main -->
        <main class=" bg-gray-50 pb-15 px-5">
        <!-- hero section -->
        <section class="container m-auto">
                <div class="hero min-h-[calc(100vh-6rem)] ">
                <div class="hero-content text-center flex-col">
                <div class="max-w-4xl">
                        <h1 class="pb-2 text-5xl md:text-7xl font-semibold bg-
gradient-to-r  from-gray-800  via-gray-700  to-gray-800  inline-block  text-
transparent bg-clip-text">Compiler Theory with Ease!</h1>
                        <p class="py-6 text-gray-500">
                        Struggling with Left Recursion, Left Factoring, or First-
Follow calculations? Compiler Forge Toolkit simplifies compiler construction
with instant, accurate outputs.
                        </p>
                        <div class="flex justify-center gap-5">
                        <a href="tools.html"><button id="btn-gs" class="btn btn-
lg bg-red-500 text-white border-red-500 hover:bg-transparent hover:text-red-
500">Get Started</button></a>
                        </div>
                </div>
                <div class="w-full h-full">
                <img class="w-full h-full object-cover" src="assets/hero 1.png"
alt="">
                </div>
                </div>
                </div>
        </section>
        <!-- steps -->
        <section class="container m-auto">
                <h3 class="text-center text-3xl font-semibold text-red-500">How
to Use</h3>
                <div class="py-10 max-w-2xl mx-auto">
                <ul class="timeline timeline-snap-icon max-md:timeline-compact
timeline-vertical">
                        <li>
                        <div class="timeline-middle">
                        <div  class="flex  items-center  justify-center  h-7  w-7
rounded-full bg-red-500 text-sm text-white font-bold">
                        01
                        </div>
                        </div>
                        <div class="timeline-start mr-5 mb-10 md:text-end">
                        <div class="text-lg font-bold">Choose Operation </div>
                        <p  class="text-gray-600">Select  the  transformation  or
analysis you need</p>
                        </div>
                        <hr />
                        </li>
                        <li>
                        <hr />
                        <div class="timeline-middle">
                        <div  class="flex  items-center  justify-center  h-7  w-7
rounded-full bg-red-500 text-sm text-white font-bold">
                                02
                        </div>
                        </div>
                        <div class="timeline-end ml-5 md:mb-10">
```

```
                        <div class="text-lg font-bold">Input Grammar</div>
                        <p class="text-gray-600">Enter your grammar rules in the
provided input field</p>
                        </div>
                        <hr />
                        </li>
                        <li>
                        <hr />
                        <div class="timeline-middle">
                        <div class="flex items-center justify-center h-7 w-7
rounded-full bg-red-500 text-sm text-white font-bold">
                                03
                        </div>
                        </div>
                        <div class="timeline-start mr-5 mb-10 md:text-end">
                        <div class="text-lg font-bold">Get Instant Results</div>
                        <p class="text-gray-600">Click the "Generate" button and
get your transformed grammar</p>
                        </div>
                        </li>
                </ul>
                </div>
        </section>


        <!-- card -->
        <section class="container m-auto my-20">
                <div class="rounded-3xl bg-gradient-to-r from-red-600 to-red-
500 text-white my-10">
                <div class="max-w-3xl px-5 py-10 flex items-center justify-
center gap-5 md:gap-30 mx-auto">
                        <div>
                        <h3 class="text-lg sm:text-3xl">No logins. No
subscriptions.
                                Just powerful compiler tools at your
fingertips.</h3>
                        </div>

                        <div>
                        <div class="">
                                <div class="p-4 rounded-full border-2 border-
red-400">
                                <div class="p-4 rounded-full border-2 border-
red-400">
                                        <div class="p-4 rounded-full border-2
border-red-400">
                                        <div class="p-4 rounded-full border-2
border-red-400">
                                                <a href="tools.html"><button
class="btn bg-white text-red-500 border-red-400 hover:bg-transparent
hover:text-white rounded-full w-12 h-12 flex items-center justify-
center">Go</button></a>
                                        </div>
                                        </div>
                                </div>
                                </div>
                        </div>
```

```
                        </div>
                </div>
                </div>
        </section>


        <!-- key features -->
        <section class="container m-auto my-30 px-15">
                <h3 class="text-center text-3xl font-semibold text-red-500">Key
Features</h3>
                <div class="text-center grid grid-cols-1 md:grid-cols-2 lg:grid-
cols-3 gap-5 py-2">
                <div class="flex flex-col items-center rounded-lg p-5 gap-3">
                        <div class="flex items-center justify-center w-10 h-10
rounded-full bg-red-500">
                        <i     class="text-white     fa-solid     fa-wand-magic-
sparkles"></i>
                        </div>
                        <h4 class="text-xl font-semibold mb-1">Left Recursion
Elimination</h4>
                        <p class="text-gray-600">Automatically transform left-
recursive grammars into equivalent non-left-recursive forms for easier
parsing.</p>
                </div>
                <div class=" flex flex-col items-center rounded-lg p-5 gap-3">
                        <div class="flex items-center justify-center w-10 h-10
rounded-full bg-red-500">
                        <i     class="text-white     fa-solid     fa-wand-magic-
sparkles"></i>
                        </div>
                        <h4     class="text-xl     font-semibold     mb-1">Left
Factoring</h4>
                        <p class="text-gray-600">Optimize grammars by factoring
out common prefixes, making them suitable for predictive parsing.</p>
                </div>
                <div class="flex flex-col items-center rounded-lg p-5 gap-3">
                        <div class="flex items-center justify-center w-10 h-10
rounded-full bg-red-500">
                        <i     class="text-white     fa-solid     fa-wand-magic-
sparkles"></i>
                        </div>
                        <h4 class="text-xl font-semibold mb-1">First and Follow
Set Calculation</h4>
                        <p class="text-gray-600">Instantly compute the First and
Follow sets of a given grammar to aid in LL(1) parsing.</p>
                </div>
                </div>
        </section>
        </main>
        <!-- footer -->
        <footer class="bg-gray-50 border-gray-200 pb-15 px-5">
        <footer class="container mx-auto footer sm:footer-horizontal rounded-
2xl bg-gray-200 text-base-content p-5 sm:p-10">
                <aside class="md:max-w-sm">
                <img src="assets/logo_header.png" alt="">
                <p class="pl-2">
                Compiler Forge Toolkit simplifies compiler construction with
instant, accurate outputs.
```

```
                </p>
                <br>
                <br>
                <small class="pl-2"> © 2025 Compiler Forge Toolkit. All Rights
Reserved.</small>
                </aside>
                <nav>
                <h6 class="font-semibold text-lg text-black">Useful Link</h6>
                <a class="link link-hover">Home</a>
                <a class="link link-hover">Features</a>
                <a class="link link-hover">Tools</a>
                <a class="link link-hover">About</a>
                <a class="link link-hover">Contact</a>
                </nav>
                <nav>
                <h6 class="font-semibold text-lg text-black">Follow Us</h6>
                <a class="link link-hover">Facebook</a>
                <a class="link link-hover">Instagram</a>
                <a class="link link-hover">LinkedIn</a>
                <a class="link link-hover">X (Twitter)</a>
                <a class="link link-hover">Github</a>
                </nav>
                <nav class="text-right">
                <form>
                <h6 class="font-semibold text-lg text-black">Subscribe to our
newsletter </h6>
                <fieldset class="w-80">
                <div class="join">
                        <input
                        type="text"
                        placeholder="Enter your email"
                        class="input input-bordered join-item" />
                        <button    class="btn    bg-red-500    text-white    join-
item">Subscribe</button>
                </div>
                </fieldset>
                <br><br><br><br><br>
                <small class="pl-2 text-right"> Made with <span class="text-red-
600">❤</span> by <strong>MarsLab</strong></small>
                </form>
                </nav>
                </footer>
        </footer>
        <script src="scripts/index.js"></script>
</body>
</html>
```

tools.html

```
<!DOCTYPE html>
<html lang="en">
<head>
	<meta charset="UTF-8">
	<meta name="viewport" content="width=device-width, initial-scale=1.0">
	<title>Tools - Compiler Forge Toolkit</title>
	<!-- favicon -->
	<link rel="icon" href="assets/fav_ico.png" type="image/x-icon">
	<!-- daisy and tailwind -->
	<link  href="https://cdn.jsdelivr.net/npm/daisyui@5"  rel="stylesheet"
type="text/css" />
	<script
src="https://cdn.jsdelivr.net/npm/@tailwindcss/browser@4"></script>
	<!-- google font -->
	<link rel="preconnect" href="https://fonts.googleapis.com">
	<link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
	<link
	href="https://fonts.googleapis.com/css2?family=Poppins:ital,wght@0,100
;0,200;0,300;0,400;0,500;0,600;0,700;0,800;0,900;1,100;1,200;1,300;1,400;1,50
0;1,600;1,700;1,800;1,900&display=swap"
	rel="stylesheet">
	<!-- font awesome -->
	<script                src="https://kit.fontawesome.com/f6fc430968.js"
crossorigin="anonymous"></script>
	<!-- custom css -->
	<style>
	.active {
		background-color: #ef4444;
		color: white;
	}
	</style>
</head>
<body class="font-[Poppins] text-gray-900">
	<!-- header -->
	<header class="bg-gray-50 border-b-2 border-gray-200 sticky top-0 z-50
py-5 h-5rem">
	<nav class="px-5 container m-auto flex justify-between items-center">
		<div>
		<a	href="index.html"><img	class="h-8	sm:h-12"
src="assets/logo_header.png" alt=""></a>
		</div>
		<!-- mid -->
		<div>
		<ul class="flex items-center gap-7 font-semibold cursor-pointer">
		<li	class="hidden	md:block"><a	href="index.html"
class="hover:text-red-600">Home</a></li>
		<li	class="hidden	md:block"><a	href="pricing.html"
class="hover:text-red-600">Pricing</a></li>
		<li id="btn-tools" class="py-1 text-lg text-red-500 border-red-
500 transition-all duration-300 hover:scale-110"><i class="fa-solid fa-wand-
magic-sparkles"></i> Tools</li>
		<li	class="hidden	md:block"><a	href="about.html"
class="hover:text-red-600">About</a></li>
		<li class="hidden md:block"><a href="contact.html" hover:text-
red-600">Contact</a></li>
```

```
                </ul>
                </div>
                <div>
                <button
                class="btn btn-sm sm:btn-md py-1 bg-red-500 text-white border-
red-500 hover:bg-transparent hover:text-red-500">Report a bug</button>
                </div>
        </nav>
        </header>
        <main class="border-gray-200 bg-gray-100 pb-15 px-5">
        <section class="container m-auto py-10 grid grid-cols-1 md:grid-cols-3
md:gap-5">
                <aside class="col-span-1 sm:pr-5 md:border-r-2 border-red-500 ">
                <div class="mb-10 flex flex-col flex-wrap justify-center gap-5">
                        <button id="lr"
                        class="rounded-xl  sm:text-lg  active  btn-tool  btn  bg-
white text-red-500 border-red-500 hover:bg-red-500 hover:text-white">Left
                        Recursion Elimination</button>
                        <button id="lf"
                        class="rounded-xl sm:text-lg btn-tool btn bg-white text-
red-500 border-red-500 hover:bg-red-500 hover:text-white">Left
                        Factoring</button>
                        <button id="ff"
                        class="rounded-xl sm:text-lg btn-tool btn bg-white text-
red-500 border-red-500 hover:bg-red-500 hover:text-white">First
                        and Follow Set Calculation</button>
                </div>
                <div class="mb-5 border-1 p-5 rounded-xl bg-white border-gray-
300">
                        <h3 class="font-semibold mb-2">Supported Grammars</h3>
                        <ul class="text-gray-500 px-7 text-sm list-disc">
                        <li>A -> A c | A a d | b d | ϵ <br>
                                (All  tokens  must  be  separated  by  space
characters)</li>
                        <li>A -> A c
                                <br>  | A a d
                                <br>  | b d
                                <br>  | ϵ
                        </li>
                        <li>S -> A a | b
                                <br>A -> A c | S d | ϵ
                        </li>
                        <li>(Copy ϵ to input if needed)</li>
                        </ul>
                </div>
                <div  class="mb-5  md:mb-0  border-1  p-5  rounded-xl  bg-white
border-gray-300">
                        <h3 class="font-semibold mb-2">Example</h3>
                        <ul id="grammar-list" class="text-gray-500 px-2 text-sm
list-disc">
                        <li class="py-2 border-b-1 border-gray-300 flex justify-
between items-center">S -> S S + | S S * | a
                                <button  class="copy-btn  btn  btn-sm  px-2  py-1
rounded text-xs text-gray-600">Copy</button>
                        </li>

                        <li class="py-2 border-b-1 border-gray-300 flex justify-
between items-center">S -> 0 S 1 | 0 1
```

```
                            <button class="copy-btn btn btn-sm px-2 py-1
rounded text-xs text-gray-600">Copy</button>
                        </li>
                        <li class="py-2 border-b-1 border-gray-300 flex justify-
between items-center">S -> ( L ) | a
                            <br>L -> L , S | S
                            <button class="copy-btn btn btn-sm px-2 py-1
rounded text-xs text-gray-600">Copy</button>
                        </li>
                        <li class="py-2 border-b-1 border-gray-300 flex justify-
between items-center">S -> a S b S | b S a S | ϵ
                            <button class="copy-btn btn btn-sm px-2 py-1
rounded text-xs text-gray-600">Copy</button>
                        </li>
                        <li class="py-2 border-b-1 border-gray-300 flex justify-
between items-center">S -> S + S | S S | ( S ) | S * | a
                            <button class="copy-btn btn btn-sm px-2 py-1
rounded text-xs text-gray-600">Copy</button>
                        </li>
                        <li class="py-2 border-gray-300 flex justify-between
items-center">S -> + S S | * S S | a
                            <button class="copy-btn btn btn-sm px-2 py-1
rounded text-xs text-gray-600">Copy</button>
                        </li>
                        </ul>
                </div>
                </aside>
                <section class="col-span-2">
                <!-- left recursion -->
                <section id="sec-lr" class="section rounded-xl">
                        <div class="px-5 pb-5 grid grid-cols-1 gap-5">
                        <div class="border-1 p-5 rounded-xl bg-white border-
gray-300">
                            <h2 class="text-xl font-bold mb-5">Input</h2>
                            <textarea id="lr-input" class="w-full h-60 p-3
border-2 border-gray-300 rounded-lg resize-none"
                            placeholder="Enter your grammar
here..."></textarea>
                            <div class="flex justify-center mt-2">
                            <button id="btn-process-lr" class="btn btn-wide
rounded-3xl bg-red-500 text-white text-lg">Process</button>
                            </div>
                        </div>
                        <div class="border-1 p-5 rounded-xl bg-white border-
gray-300">
                            <h2 class="text-xl font-bold mb-5">Output</h2>
                            <div class="w-full h-75 p-3 border-2 border-gray-
300 rounded-lg">
                            <p id="lr-output" class="text-gray-500">Output
will be displayed here...</p>
                            </div>
                        </div>
                        </div>
                </section>
                <!-- left factoring -->
                <section id="sec-lf" class="hidden section rounded-xl">
                        <div class="px-5 pb-5 grid grid-cols-1 gap-5">
```

```
                    <div class="border-1 p-5 rounded-xl bg-white border-
gray-300">
                            <h2 class="text-xl font-bold mb-5">Input</h2>
                            <textarea id="lf-input" class="w-full h-60 p-3
border-2 border-gray-300 rounded-lg resize-none"
                            placeholder="Enter your grammar
here..."></textarea>
                            <div class="flex justify-center mt-2">
                            <button id="btn-process-lf" class="btn btn-wide
rounded-3xl bg-red-500 text-white text-lg">Process</button>
                            </div>
                    </div>
                    <div class="border-1 p-5 rounded-xl bg-white border-
gray-300">
                            <h2 class="text-xl font-bold mb-5">Output</h2>
                            <div class="w-full h-75 p-3 border-2 border-gray-
300 rounded-lg">
                            <p id="lf-output" class="text-gray-500">Output
will be displayed here...</p>
                            </div>
                            <!-- <textarea class="w-full h-40 p-2 border-2
border-gray-300 rounded-lg resize-none" placeholder="Enter your input
here..."></textarea> -->
                    </div>
                    </div>
            </section>
            <!-- first and follow -->
            <section id="sec-ff" class="hidden section rounded-xl">
                    <div class="px-5 pb-5 grid grid-cols-1 gap-5">
                    <div class="border-1 p-5 rounded-xl bg-white border-
gray-300">
                            <h2 class="text-xl font-bold mb-5">Input</h2>
                            <textarea id="ff-input" class="w-full h-60 p-3
border-2 border-gray-300 rounded-lg resize-none"
                            placeholder="Enter your grammar
here..."></textarea>
                            <div class="flex justify-center mt-2">
                            <button id="btn-process-ff" class="btn btn-wide
rounded-3xl bg-red-500 text-white text-lg">Process</button>
                            </div>
                    </div>
                    <div class="border-1 p-5 rounded-xl bg-white border-
gray-300">
                            <h2 class="text-xl font-bold mb-5">Output</h2>
                            <div class="overflow-auto w-full h-75 p-3 border-
2 border-gray-300 rounded-lg">
                            <div id="ff-output">
                                    <table
                                    class="text-sm md:text-lg text-black
overflow-auto w-full rounded-box border-collapse border border-gray-300">
                                    <thead>
                                    <tr>
                                            <th class='border border-gray-300
px-4 py-2'>Non-Terminal</th>
                                            <th class='border border-gray-300
px-4 py-2'>First</th>
                                            <th class='border border-gray-300
px-4 py-2'>Follow</th>
```

```
                                        </tr>
                                        </thead>
                                        <tbody>
                                        </tbody>
                                </table>
                                </div>
                                </div>
                                <!-- <textarea class="w-full h-40 p-2 border-2
border-gray-300   rounded-lg   resize-none"   placeholder="Enter   your   input
here..."></textarea> -->
                        </div>
                        </div>
                </section>
                </section>
        </section>
        <!-- first and follow -->
        </main>
        <!-- footer -->
        <footer class="bg-gray-100  pb-15 px-5">
        <footer class="container mx-auto footer sm:footer-horizontal rounded-
2xl bg-gray-200 text-base-content p-5 sm:p-10">
                <aside class="md:max-w-sm">
                <img src="assets/logo_header.png" alt="">
                <p class="pl-2">
                Compiler Forge Toolkit simplifies compiler construction with
instant, accurate outputs.
                </p>
                <br>
                <br>
                <small class="pl-2"> © 2025 Compiler Forge Toolkit. All Rights
Reserved.</small>
                </aside>
                <nav>
                <h6 class="font-semibold text-lg text-black">Useful Link</h6>
                <a class="link link-hover">Home</a>
                <a class="link link-hover">Features</a>
                <a class="link link-hover">Tools</a>
                <a class="link link-hover">About</a>
                <a class="link link-hover">Contact</a>
                </nav>
                <nav>
                <h6 class="font-semibold text-lg text-black">Follow Us</h6>
                <a class="link link-hover">Facebook</a>
                <a class="link link-hover">Instagram</a>
                <a class="link link-hover">LinkedIn</a>
                <a class="link link-hover">X (Twitter)</a>
                <a class="link link-hover">Github</a>
                </nav>
                <nav class="text-right">
                <form>
                <h6 class="font-semibold text-lg text-black">Subscribe to our
newsletter </h6>
                <fieldset class="w-80">
                <div class="join">
                        <input
                        type="text"
                        placeholder="Enter your email"
                        class="input input-bordered join-item" />
```

```
                    <button    class="btn    bg-red-500    text-white    join-
item">Subscribe</button>
                </div>
                </fieldset>
                <br><br><br><br><br>
                <small class="pl-2 text-right"> Made with <span class="text-red-
600">♥</span> by <strong>MarsLab</strong></small>
                </form>
                </nav>
                </footer>
        </footer>
        <script src="scripts/tools.js"></script>
        <script src="scripts/basic.js"></script>
        <script src="scripts/lf.js"></script>
        <script src="scripts/lr.js"></script>
        <script src="scripts/ff.js"></script>
</body>
</html>
```

about.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>About - Compiler Forge Toolkit</title>
  <!-- favicon -->
  <link rel="icon" href="assets/fav_ico.png" type="image/x-icon">
  <!-- daisy and tailwind -->
  <link     href="https://cdn.jsdelivr.net/npm/daisyui@5"     rel="stylesheet"
type="text/css" />
  <script src="https://cdn.jsdelivr.net/npm/@tailwindcss/browser@4"></script>
  <!-- google font -->
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <link
href="https://fonts.googleapis.com/css2?family=Poppins:ital,wght@0,100;0,200;
0,300;0,400;0,500;0,600;0,700;0,800;0,900;1,100;1,200;1,300;1,400;1,500;1,600
;1,700;1,800;1,900&display=swap" rel="stylesheet">
  <!-- font awesome -->
  <script                      src="https://kit.fontawesome.com/f6fc430968.js"
crossorigin="anonymous"></script>
</head>
<body class="font-[Poppins] text-gray-900">
        <!-- header -->
        <header class="bg-gray-50 border-b-2 border-gray-200 sticky top-0 z-50
py-5 h-5rem">
        <nav class="px-5 container m-auto flex justify-between items-center">
                <div>
                <a       href="index.html"><img       class="h-8       sm:h-12"
src="assets/logo_header.png" alt=""></a>
                </div>

                <!-- mid -->
                <div>
```

```
            <ul class="flex items-center gap-7 font-semibold cursor-pointer">
                <li      class="hidden      md:block"><a      href="index.html"
class="hover:text-red-600">Home</a></li>
                <li     class="hidden     md:block"><a     href="pricing.html"
class="hover:text-red-600">Pricing</a></li>
                <li id="btn-tools" class="py-1 text-lg text-red-500 border-red-
500 transition-all duration-300 hover:scale-110"><i class="fa-solid fa-wand-
magic-sparkles"></i> Tools</li>
                <li class="hidden md:block"><a href="about.html" class="border-
b-2 border-red-500 hover:text-red-600">About</a></li>
                <li  class="hidden  md:block"><a  href="contact.html"  class="
hover:text-red-600">Contact</a></li>
                </ul>
                </div>
                <div>
                <button
                class="btn btn-sm sm:btn-md py-1 bg-red-500 text-white border-
red-500 hover:bg-transparent hover:text-red-500">Report a bug</button>
                </div>
        </nav>
        </header>
        <!-- main -->
        <main class=" bg-gray-50 pb-15 px-5">
        <!-- hero section -->
        <section class="container m-auto">
                <div class="hero min-h-[calc(100vh-6rem)] ">
                <div class="hero-content text-center flex-col">
                <div class="max-w-4xl">
                        <h1 class="pb-2 text-5xl md:text-7xl font-semibold bg-
gradient-to-r  from-gray-800  via-gray-700  to-gray-800  inline-block  text-
transparent bg-clip-text">Under Development</h1>
                        <p class="py-6 text-gray-500">
                        This page is currently under development! We're working
hard to bring you the best experience.
                        </p>
                        <div class="flex justify-center gap-5">
                        <a href="index.html"><button id="btn-gs" class="btn btn-
lg bg-red-500 text-white border-red-500 hover:bg-transparent hover:text-red-
500">Return to home</button></a>
                        </div>
                </div>
                </div>
                </div>
        </section>
        <!-- card -->
        <section class="container m-auto my-20">
                <div class="rounded-3xl bg-gradient-to-r from-red-600 to-red-
500 text-white my-10">
                <div  class="max-w-3xl  px-5  py-10  flex  items-center  justify-
center gap-5 md:gap-30 mx-auto">
                        <div>
                        <h3 class="text-3xl">No logins. No subscriptions.
                                Just   powerful   compiler   tools   at   your
fingertips.</h3>
                        </div>

                        <div>
                        <div class="">
```

```
                            <div class="p-4 rounded-full border-2 border-
red-400">
                            <div class="p-4 rounded-full border-2 border-
red-400">
                                    <div class="p-4 rounded-full border-2
border-red-400">
                                    <div class="p-4 rounded-full border-2
border-red-400">
                                            <a href="tools.html"><button
class="btn bg-white text-red-500 border-red-400 hover:bg-transparent
hover:text-white rounded-full w-12 h-12 flex items-center justify-
center">Go</button></a>
                                    </div>
                                    </div>
                            </div>
                            </div>
                        </div>
                        </div>

                </div>
                </div>
        </section>
        </main>
        <!-- footer -->
        <footer class="bg-gray-50 border-gray-200 pb-15 px-5">
        <footer class="container mx-auto footer sm:footer-horizontal rounded-
2xl bg-gray-200 text-base-content p-5 sm:p-10">
                <aside class="md:max-w-sm">
                <img src="assets/logo_header.png" alt="">
                <p class="pl-2">
                Compiler Forge Toolkit simplifies compiler construction with
instant, accurate outputs.
                </p>
                <br>
                <br>
                <small class="pl-2"> © 2025 Compiler Forge Toolkit. All Rights
Reserved.</small>
                </aside>
                <nav>
                <h6 class="font-semibold text-lg text-black">Useful Link</h6>
                <a class="link link-hover">Home</a>
                <a class="link link-hover">Features</a>
                <a class="link link-hover">Tools</a>
                <a class="link link-hover">About</a>
                <a class="link link-hover">Contact</a>
                </nav>
                <nav>
                <h6 class="font-semibold text-lg text-black">Follow Us</h6>
                <a class="link link-hover">Facebook</a>
                <a class="link link-hover">Instagram</a>
                <a class="link link-hover">LinkedIn</a>
                <a class="link link-hover">X (Twitter)</a>
                <a class="link link-hover">Github</a>
                </nav>
                <nav class="text-right">
                <form>
                <h6 class="font-semibold text-lg text-black">Subscribe to our
newsletter </h6>
```

```
                <fieldset class="w-80">
                <div class="join">
                        <input
                        type="text"
                        placeholder="Enter your email"
                        class="input input-bordered join-item" />
                        <button    class="btn    bg-red-500    text-white    join-
item">Subscribe</button>
                </div>
                </fieldset>
                <br><br><br><br><br>
                <small class="pl-2 text-right"> Made with <span class="text-red-
600">♥</span> by <strong>MarsLab</strong></small>
                </form>
                </nav>
                </footer>
        </footer>
        <script src="scripts/index.js"></script>
</body>
</html>
```

contact.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Contact - Compiler Forge Toolkit</title>
  <!-- favicon -->
  <link rel="icon" href="assets/fav_ico.png" type="image/x-icon">
  <!-- daisy and tailwind -->
  <link     href="https://cdn.jsdelivr.net/npm/daisyui@5"     rel="stylesheet"
type="text/css" />
  <script src="https://cdn.jsdelivr.net/npm/@tailwindcss/browser@4"></script>
  <!-- google font -->
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <link
href="https://fonts.googleapis.com/css2?family=Poppins:ital,wght@0,100;0,200;
0,300;0,400;0,500;0,600;0,700;0,800;0,900;1,100;1,200;1,300;1,400;1,500;1,600
;1,700;1,800;1,900&display=swap" rel="stylesheet">
  <!-- font awesome -->
  <script                      src="https://kit.fontawesome.com/f6fc430968.js"
crossorigin="anonymous"></script>
</head>
<body class="font-[Poppins] text-gray-900">
        <!-- header -->
        <header class="bg-gray-50 border-b-2 border-gray-200 sticky top-0 z-50
py-5 h-5rem">
        <nav class="px-5 container m-auto flex justify-between items-center">
                <div>
                <a       href="index.html"><img       class="h-8       sm:h-12"
src="assets/logo_header.png" alt=""></a>
                </div>
```

```
                <!-- mid -->
                <div>
                <ul class="flex items-center gap-7 font-semibold cursor-pointer">
                <li      class="hidden      md:block"><a      href="index.html"
class="hover:text-red-600">Home</a></li>
                <li     class="hidden     md:block"><a     href="pricing.html"
class="hover:text-red-600">Pricing</a></li>
                <li id="btn-tools" class="py-1 text-lg text-red-500 border-red-
500 transition-all duration-300 hover:scale-110"><i class="fa-solid fa-wand-
magic-sparkles"></i> Tools</li>
                <li      class="hidden      md:block"><a      href="about.html"
class="hover:text-red-600">About</a></li>
                <li     class="hidden     md:block"><a     href="contact.html"
class="border-b-2 border-red-500 hover:text-red-600">Contact</a></li>
                </ul>
                </div>
                <div>
                <button
                class="btn btn-sm sm:btn-md py-1 bg-red-500 text-white border-
red-500 hover:bg-transparent hover:text-red-500">Report a bug</button>
                </div>
        </nav>
        </header>
        <!-- main -->
        <main class=" bg-gray-50 pb-15 px-5">
        <!-- hero section -->
        <section class="container m-auto">
                <div class="hero min-h-[calc(100vh-6rem)] ">
                <div class="hero-content text-center flex-col">
                <div class="max-w-4xl">
                        <h1 class="pb-2 text-5xl md:text-7xl font-semibold bg-
gradient-to-r  from-gray-800  via-gray-700  to-gray-800  inline-block  text-
transparent bg-clip-text">Contact Us</h1>
                        <p class="pt-6 pb-2 text-lg text-gray-600">
                        <i       class="fa-solid       fa-envelope"></i>       <a
href="mailto:souravgarodia111@gmail.com">souravgarodia111@gmail.com</a>
                        </p>
                        <p  class="pb-6  text-lg  text-gray-600"><i  class="fa-
brands fa-github"></i> <a href="https://github.com/garodiaa">garodiaa</a></p>
                        <div class="flex justify-center gap-5">
                        <a href="index.html"><button id="btn-gs" class="btn btn-
lg bg-red-500 text-white border-red-500 hover:bg-transparent hover:text-red-
500">Return to home</button></a>
                        </div>
                </div>
                </div>
                </div>
        </section>
        <!-- card -->
        <section class="container m-auto my-20">
                <div class="rounded-3xl bg-gradient-to-r from-red-600 to-red-
500 text-white my-10">
                <div class="max-w-3xl  px-5  py-10  flex  items-center  justify-
center gap-5 md:gap-30 mx-auto">
                        <div>
                        <h3 class="text-3xl">No logins. No subscriptions.
                                Just   powerful   compiler   tools   at   your
fingertips.</h3>
```

```
                    </div>

                    <div>
                    <div class="">
                            <div class="p-4 rounded-full border-2 border-
red-400">
                            <div class="p-4 rounded-full border-2 border-
red-400">
                                    <div class="p-4 rounded-full border-2
border-red-400">
                                    <div class="p-4 rounded-full border-2
border-red-400">
                                            <a href="tools.html"><button
class="btn bg-white text-red-500 border-red-400 hover:bg-transparent
hover:text-white rounded-full w-12 h-12 flex items-center justify-
center">Go</button></a>
                                    </div>
                                    </div>
                            </div>
                            </div>
                    </div>
                    </div>

            </div>
            </div>
        </section>
        </main>
        <!-- footer -->
        <footer class="bg-gray-50 border-gray-200 pb-15 px-5">
        <footer class="container mx-auto footer sm:footer-horizontal rounded-
2xl bg-gray-200 text-base-content p-5 sm:p-10">
            <aside class="md:max-w-sm">
            <img src="assets/logo_header.png" alt="">
            <p class="pl-2">
            Compiler Forge Toolkit simplifies compiler construction with
instant, accurate outputs.
            </p>
            <br>
            <br>
            <small class="pl-2"> © 2025 Compiler Forge Toolkit. All Rights
Reserved.</small>
            </aside>
            <nav>
            <h6 class="font-semibold text-lg text-black">Useful Link</h6>
            <a class="link link-hover">Home</a>
            <a class="link link-hover">Features</a>
            <a class="link link-hover">Tools</a>
            <a class="link link-hover">About</a>
            <a class="link link-hover">Contact</a>
            </nav>
            <nav>
            <h6 class="font-semibold text-lg text-black">Follow Us</h6>
            <a class="link link-hover">Facebook</a>
            <a class="link link-hover">Instagram</a>
            <a class="link link-hover">LinkedIn</a>
            <a class="link link-hover">X (Twitter)</a>
            <a class="link link-hover">Github</a>
            </nav>
```

```
                <nav class="text-right">
                <form>
                <h6 class="font-semibold text-lg text-black">Subscribe to our
newsletter </h6>
                <fieldset class="w-80">
                <div class="join">
                        <input
                        type="text"
                        placeholder="Enter your email"
                        class="input input-bordered join-item" />
                        <button    class="btn    bg-red-500    text-white    join-
item">Subscribe</button>
                </div>
                </fieldset>
                <br><br><br><br><br>
                <small class="pl-2 text-right"> Made with <span class="text-red-
600">♥</span> by <strong>MarsLab</strong></small>
                </form>
                </nav>
                </footer>
        </footer>
        <script src="scripts/index.js"></script>
</body>
</html>
```

pricing.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Pricing - Compiler Forge Toolkit</title>
  <!-- favicon -->
  <link rel="icon" href="assets/fav_ico.png" type="image/x-icon">
  <!-- daisy and tailwind -->
  <link     href="https://cdn.jsdelivr.net/npm/daisyui@5"     rel="stylesheet"
type="text/css" />
  <script src="https://cdn.jsdelivr.net/npm/@tailwindcss/browser@4"></script>
  <!-- google font -->
  <link rel="preconnect" href="https://fonts.googleapis.com">
  <link rel="preconnect" href="https://fonts.gstatic.com" crossorigin>
  <link
href="https://fonts.googleapis.com/css2?family=Poppins:ital,wght@0,100;0,200;
0,300;0,400;0,500;0,600;0,700;0,800;0,900;1,100;1,200;1,300;1,400;1,500;1,600
;1,700;1,800;1,900&display=swap" rel="stylesheet">
  <!-- font awesome -->
  <script                      src="https://kit.fontawesome.com/f6fc430968.js"
crossorigin="anonymous"></script>
</head>
<body class="font-[Poppins] text-gray-900">
        <!-- header -->
        <header class="bg-gray-50 border-b-2 border-gray-200 sticky top-0 z-50
py-5 h-5rem">
        <nav class="px-5 container m-auto flex justify-between items-center">
```

```
                <div>
                <a       href="index.html"><img       class="h-8       sm:h-12"
src="assets/logo_header.png" alt=""></a>
                </div>
                <!-- mid -->
                <div>
               <ul class="flex items-center gap-7 font-semibold cursor-pointer">
                <li      class="hidden      md:block"><a      href="index.html"
class="hover:text-red-600">Home</a></li>
                <li     class="hidden     md:block"><a     href="pricing.html"
class="border-b-2 border-red-500 hover:text-red-600">Pricing</a></li>
                <li id="btn-tools" class="py-1 text-lg text-red-500 border-red-
500 transition-all duration-300 hover:scale-110"><i class="fa-solid fa-wand-
magic-sparkles"></i> Tools</li>
                <li      class="hidden      md:block"><a      href="about.html"
class="hover:text-red-600">About</a></li>
                <li  class="hidden  md:block"><a  href="contact.html"  class="
hover:text-red-600">Contact</a></li>
                </ul>
                </div>
                <div>
                <button
                class="btn btn-sm sm:btn-md py-1 bg-red-500 text-white border-
red-500 hover:bg-transparent hover:text-red-500">Report a bug</button>
                </div>
        </nav>
        </header>
        <!-- main -->
        <main class=" bg-gray-50 pb-15 px-5">
        <!-- hero section -->
        <section class="container m-auto">
                <div class="hero min-h-[calc(100vh-6rem)] ">
                <div class="hero-content text-center flex-col">
                <div class="max-w-4xl">
                        <h1 class="pb-2 text-5xl md:text-7xl font-semibold bg-
gradient-to-r  from-gray-800  via-gray-700  to-gray-800  inline-block  text-
transparent bg-clip-text">Under Development</h1>
                        <p class="py-6 text-gray-500">
                        This page is currently under development! We're working
hard to bring you the best experience.
                        </p>
                        <div class="flex justify-center gap-5">
                        <a href="index.html"><button id="btn-gs" class="btn btn-
lg bg-red-500 text-white border-red-500 hover:bg-transparent hover:text-red-
500">Return to home</button></a>
                        </div>
                </div>
                </div>
                </div>
        </section>
        <!-- card -->
        <section class="container m-auto my-20">
                <div class="rounded-3xl bg-gradient-to-r from-red-600 to-red-
500 text-white my-10">
                <div class="max-w-3xl px-5 py-10 flex items-center justify-
center gap-5 md:gap-30 mx-auto">
                        <div>
                        <h3 class="text-3xl">No logins. No subscriptions.
```

```
                        Just powerful compiler tools at your
fingertips.</h3>
                    </div>
                    <div>
                    <div class="">
                        <div class="p-4 rounded-full border-2 border-
red-400">
                        <div class="p-4 rounded-full border-2 border-
red-400">
                            <div class="p-4 rounded-full border-2
border-red-400">
                            <div class="p-4 rounded-full border-2
border-red-400">
                                <a href="tools.html"><button
class="btn bg-white text-red-500 border-red-400 hover:bg-transparent
hover:text-white rounded-full w-12 h-12 flex items-center justify-
center">Go</button></a>
                            </div>
                            </div>
                        </div>
                        </div>
                    </div>
                    </div>

            </div>
            </div>
        </section>
        </main>
        <!-- footer -->
        <footer class="bg-gray-50 border-gray-200 pb-15 px-5">
        <footer class="container mx-auto footer sm:footer-horizontal rounded-
2xl bg-gray-200 text-base-content p-5 sm:p-10">
            <aside class="md:max-w-sm">
            <img src="assets/logo_header.png" alt="">
            <p class="pl-2">
            Compiler Forge Toolkit simplifies compiler construction with
instant, accurate outputs.
            </p>
            <br>
            <br>
            <small class="pl-2"> © 2025 Compiler Forge Toolkit. All Rights
Reserved.</small>
            </aside>
            <nav>
            <h6 class="font-semibold text-lg text-black">Useful Link</h6>
            <a class="link link-hover">Home</a>
            <a class="link link-hover">Features</a>
            <a class="link link-hover">Tools</a>
            <a class="link link-hover">About</a>
            <a class="link link-hover">Contact</a>
            </nav>
            <nav>
            <h6 class="font-semibold text-lg text-black">Follow Us</h6>
            <a class="link link-hover">Facebook</a>
            <a class="link link-hover">Instagram</a>
            <a class="link link-hover">LinkedIn</a>
            <a class="link link-hover">X (Twitter)</a>
            <a class="link link-hover">Github</a>
```

```
                </nav>
                <nav class="text-right">
                <form>
                <h6 class="font-semibold text-lg text-black">Subscribe to our
newsletter </h6>
                <fieldset class="w-80">
                <div class="join">
                        <input
                        type="text"
                        placeholder="Enter your email"
                        class="input input-bordered join-item" />
                        <button    class="btn    bg-red-500    text-white    join-
item">Subscribe</button>
                </div>
                </fieldset>
                <br><br><br><br><br>
                <small class="pl-2 text-right"> Made with <span class="text-red-
600">♥</span> by <strong>MarsLab</strong></small>
                </form>
                </nav>
                </footer>
        </footer>
        <script src="scripts/index.js"></script>
</body>
</html>
```

index.js

```
document.getElementById("btn-tools").addEventListener
("click", () => {
        window.location.href = "tools.html";
}
);
document.querySelectorAll(".btn").forEach((button) => {
        button.addEventListener("click", () => {
        console.log("Button clicked:", button.innerText);
        const buttonId = button.getAttribute("id");
        console.log("Button ID:", buttonId);
        });
}
);
```

basic.js

```
document.querySelectorAll(".copy-btn").forEach(button => {
        button.addEventListener("click", function () {
        const    textToCopy    =    this.parentElement.innerText.replace("Copy",
"").trim(); // Remove "Copy" text
        navigator.clipboard.writeText(textToCopy).then(() => {
                // alert("Copied: " + textToCopy);
        }).catch(err => console.error("Failed to copy:", err));
        });
});
/**
```

```
 * Format the grammar object back to a string with productions on separate lines
 * @param {Object} grammar - The structured grammar object
 * @return {string} - Formatted grammar as a string
 */
function formatGrammar(grammar) {
        let result = '';

        for (const nonTerminal in grammar) {
        const productions = grammar[nonTerminal];

        if (productions.length > 0) {
                result += nonTerminal + ' -> ' + productions[0] + '\n';

                for (let i = 1; i < productions.length; i++) {
                result += '    | ' + productions[i] + '\n';
                }
        } else {
                result += nonTerminal + ' -> \n';
        }
        // Add an extra newline between non-terminals, except for the last one
        if              (Object.keys(grammar).indexOf(nonTerminal)              <
Object.keys(grammar).length - 1) {
                result += '\n';
        }
        }
        return result.trim();
}
function orderProductions(productions, nonTerminals) {
        // Create three arrays for different types of productions
        const nonTerminalProds = [];
        const terminalProds = [];
        const epsilonProds = [];
        for (const prod of productions) {
        if (prod === 'ϵ') {
                epsilonProds.push(prod);
        } else {
                const firstToken = prod.split(' ')[0];
                if (nonTerminals.includes(firstToken)) {
                nonTerminalProds.push(prod);
                } else {
                terminalProds.push(prod);
                }
        }
        }
        // Combine the arrays in the desired order
        return [...nonTerminalProds, ...terminalProds, ...epsilonProds];
}
/**
 * Group productions by their common prefixes
 * @param {Array} productions - Array of productions
 * @return {Object} - Object with prefixes as keys and arrays of productions as
values
 */
function groupByPrefix(productions) {
        const prefixGroups = {};
        for (const prod of productions) {
        // Skip epsilon productions when looking for prefixes
        if (prod === 'ϵ') {
```

```
                if (!prefixGroups['']) {
                prefixGroups[''] = [];
                }
                prefixGroups[''].push(prod);
                continue;
        }
        // Get the first token of the production
        const tokens = prod.split(' ');
        const firstToken = tokens[0];
        // Find the longest common prefix with existing groups
        let longestPrefix = '';
        let longestPrefixLength = 0;
        for (const prefix in prefixGroups) {
                if (prefix === '') continue;
                // Check if this production starts with the current prefix
                if (prod.startsWith(prefix)) {
                // Update if this is a longer prefix
                if (prefix.length > longestPrefixLength) {
                        longestPrefix = prefix;
                        longestPrefixLength = prefix.length;
                }
                }
        }
        // If we found a matching prefix group, add to it
        if (longestPrefix !== '') {
                prefixGroups[longestPrefix].push(prod);
        }
        // Otherwise, create a new group with the first token as prefix
        else {
                if (!prefixGroups[firstToken]) {
                prefixGroups[firstToken] = [];
                }
                prefixGroups[firstToken].push(prod);
        }
        }
        return prefixGroups;
}
/**
 * Parse a grammar string into a structured object
 * @param {string} grammar - The grammar as a string
 * @return {Object} - Object with non-terminals as keys and arrays of
productions as values
 */
function parseGrammar(grammar) {
        const parsedGrammar = {};
        // Split the grammar into lines and process each line
        const lines = grammar.trim().split("\n");
        let currentNonTerminal = null;
        for (const line of lines) {
        const trimmedLine = line.trim();
        // Skip empty lines
        if (trimmedLine === "") continue;
        // Check if line defines a new non-terminal
        if (trimmedLine.includes("->")) {
        const parts = trimmedLine.split("->");
        currentNonTerminal = parts[0].trim();
        let productions = parts[1].trim();
        parsedGrammar[currentNonTerminal] = [];
```

```
        if (productions.includes("|")) {
        const prods = productions.split("|").map((p) => p.trim());
        parsedGrammar[currentNonTerminal].push(...prods);
        } else {
        parsedGrammar[currentNonTerminal].push(productions);
        }
        }
        // If line continues with productions for the current non-terminal
        else if (trimmedLine.startsWith("|") && currentNonTerminal) {
        const production = trimmedLine.substring(1).trim();
        parsedGrammar[currentNonTerminal].push(production);
        }
        }

        return parsedGrammar;
  }
```

ff.js

```
function computeFirstAndFollowSets(grammarStr) {
        // Parse the grammar and identify the start symbol
        const productions = parseGrammar(grammarStr);
        const nonTerminals = Object.keys(productions);
        const start = nonTerminals[0]; // Assuming the first non-terminal is
the start symbol
        // Initialize sets
        const first = {};
        const follow = {};
        // Initialize empty sets for all non-terminals
        nonTerminals.forEach(nt => {
        first[nt] = new Set();
        follow[nt] = new Set();
        });
        // Special case: Add $ to follow set of start symbol
        follow[start].add('$');
        // Helper function to determine if a symbol is a terminal
        function isTerminal(symbol) {
        return !nonTerminals.includes(symbol);
        }
        // Track symbols being computed to avoid infinite recursion
        const computing = new Set();
        // Function to compute First set of a symbol
        function computeFirst(symbol) {
        // If symbol is epsilon
        if (symbol === 'ε' || symbol === 'ε') {
        return new Set(['ε']);
        }
        // If symbol is a terminal
        if (isTerminal(symbol)) {
        return new Set([symbol]);
        }

        // If we're already computing this symbol's first set, return what we
have
        // to break the recursion
```

```
if (computing.has(symbol)) {
return new Set(first[symbol]);
}
// If we've already computed this non-terminal's first set
if (first[symbol].size > 0) {
return new Set(first[symbol]);
}
// Mark this symbol as being computed
computing.add(symbol);
// Compute First set for each production
productions[symbol].forEach(production => {
if (production === 'ϵ' || production === 'ε') {
// If production is epsilon, add it to First set
first[symbol].add('ε');
} else {
// Split production into symbols (assuming they're space-separated)
const symbols = production.split(' ').filter(Boolean);
// For each symbol in the production
let allDeriveEpsilon = true;
for (let i = 0; i < symbols.length; i++) {
        const currentSymbol = symbols[i];
        const currentFirst = computeFirst(currentSymbol);
        // Add all symbols from currentFirst except epsilon
        currentFirst.forEach(s => {
        if (s !== 'ε') {
        first[symbol].add(s);
        }
        });
        // If current symbol doesn't derive epsilon, stop here
        if (!currentFirst.has('ε')) {
        allDeriveEpsilon = false;
        break;
        }
        // If this is the last symbol and all symbols derive epsilon
        if (i === symbols.length - 1 && allDeriveEpsilon) {
        first[symbol].add('ε');
        }
}
}
});

// Remove this symbol from the computing set
computing.delete(symbol);
return new Set(first[symbol]);
}
// First compute all First sets
nonTerminals.forEach(nt => {
computeFirst(nt);
});
// Function to compute First set of a string of symbols
function computeFirstOfString(symbolsArray) {
if (symbolsArray.length === 0) {
return new Set(['ε']);
}
const result = new Set();
let allDeriveEpsilon = true;
for (let i = 0; i < symbolsArray.length; i++) {
const currentSymbol = symbolsArray[i];
```

```
        let currentFirst;
        if (currentSymbol === 'ϵ' || currentSymbol === 'ε') {
        currentFirst = new Set(['ε']);
        } else if (isTerminal(currentSymbol)) {
        currentFirst = new Set([currentSymbol]);
        } else {
        currentFirst = new Set(first[currentSymbol]);
        }
        // Add all symbols from currentFirst except epsilon
        currentFirst.forEach(s => {
        if (s !== 'ε') {
                result.add(s);
        }
        });
        // If current symbol doesn't derive epsilon, stop here
        if (!currentFirst.has('ε')) {
        allDeriveEpsilon = false;
        break;
        }
        // If this is the last symbol and all symbols derive epsilon
        if (i === symbolsArray.length - 1 && allDeriveEpsilon) {
        result.add('ε');
        }
        }
        return result;
        }

        // Now compute Follow sets - limit iterations to prevent infinite loops
        let followSetChanged = true;
        let iterations = 0;
        const MAX_ITERATIONS = 100; // Reasonable limit to prevent infinite
loops

        while (followSetChanged && iterations < MAX_ITERATIONS) {
        followSetChanged = false;
        iterations++;

        nonTerminals.forEach(nt => {
        // For each production of each non-terminal
        nonTerminals.forEach(lhs => {
        productions[lhs].forEach(production => {
                if (production === 'ϵ' || production === 'ε') return;

                // Split production into symbols
                const symbols = production.split(' ').filter(Boolean);

                // Find all occurrences of nt in the production
                for (let i = 0; i < symbols.length; i++) {
                if (symbols[i] === nt) {
                // Get the First set of the suffix after nt
                const suffix = symbols.slice(i + 1);
                const suffixFirst = computeFirstOfString(suffix);

                // Add all elements from suffixFirst except epsilon to Follow(nt)
                const oldSize = follow[nt].size;

                suffixFirst.forEach(s => {
                if (s !== 'ε') {
```

```
                        follow[nt].add(s);
                }
                });

                // If suffix can derive epsilon or is empty, add all elements
from Follow(lhs) to Follow(nt)
                if (suffixFirst.has('ε') || suffix.length === 0) {
                follow[lhs].forEach(s => {
                        follow[nt].add(s);
                });
                }

                // Check if Follow(nt) changed
                if (follow[nt].size > oldSize) {
                followSetChanged = true;
                }
                }
                }
        });
        });
        });
        }

        // Convert Set objects to arrays for cleaner output
        const result = {
        first: {},
        follow: {},
        grammar: { start, productions } // Include the parsed grammar for
reference
        };

        nonTerminals.forEach(nt => {
        result.first[nt] = Array.from(first[nt]);
        result.follow[nt] = Array.from(follow[nt]);
        });

        return result;
  }
```

lf.js

```
/**
 * Function to eliminate left factoring from a context-free grammar
 * @param {string} grammar - Grammar rules as a string
 * @return {string} - Grammar with left factoring eliminated
 */
function eliminateLeftFactoring(grammar) {
        // Parse the grammar into a structured format
        const parsedGrammar = parseGrammar(grammar);

        // Get all non-terminals in the grammar
        const nonTerminals = Object.keys(parsedGrammar);

        // Process each non-terminal
        const resultGrammar = {};
```

```
	const nonTerminalPrimes = {}; // Track number of primes for each non-
terminal

	for (const nonTerminal in parsedGrammar) {
	// Initialize prime count for this non-terminal
	nonTerminalPrimes[nonTerminal] = 0;

	// Apply left factoring to this non-terminal
	const { newProductions, newNonTerminals } = applyLeftFactoring(
		nonTerminal,
		parsedGrammar[nonTerminal],
		nonTerminalPrimes,
		nonTerminals
	);

	// Add the processed productions to the result
	resultGrammar[nonTerminal]	=	orderProductions(newProductions,
nonTerminals);

	// Add any new non-terminals created during left factoring
	for (const newNT in newNonTerminals) {
		nonTerminals.push(newNT);
		resultGrammar[newNT] = orderProductions(newNonTerminals[newNT],
nonTerminals);
	}
	}

	// Convert the processed grammar back to string format
	return formatGrammar(resultGrammar);
}


/**
 * Apply left factoring to a set of productions for a non-terminal
 * @param {string} nonTerminal - The non-terminal being processed
 * @param {Array} productions - Array of productions for the non-terminal
 * @param {Object} primeCount - Object tracking prime counts for non-terminals
 * @param {Array} allNonTerminals - List of all non-terminals in the grammar
 * @return {Object} - Object containing new productions and any new non-
terminals
 */
function	applyLeftFactoring(nonTerminal,	productions,	primeCount,
allNonTerminals) {
	let newProductions = [...productions];
	const newNonTerminals = {};
	let changed = true;

	// Continue until no more factoring can be done
	while (changed) {
	changed = false;

	// Group productions by their common prefixes
	const prefixGroups = groupByPrefix(newProductions);

	// Process each group with a common prefix
	for (const prefix in prefixGroups) {
		// Skip if there's only one production with this prefix or if
prefix is empty
```

```
            if (prefixGroups[prefix].length <= 1 || prefix === '') continue;

            changed = true;

            // Create a new non-terminal with prime notation
            primeCount[nonTerminal]++;
            const primes = "'".repeat(primeCount[nonTerminal]);
            const newNT = `${nonTerminal}${primes}`;
            allNonTerminals.push(newNT);

            // Extract the suffixes after the common prefix
            const suffixes = prefixGroups[prefix].map(prod => {
            const restOfProduction = prod.substring(prefix.length).trim();
            return restOfProduction === '' ? 'ϵ' : restOfProduction;
            });

            // Remove the original productions from the list
            newProductions = newProductions.filter(prod =>
            !prefixGroups[prefix].includes(prod)
            );

            // Add the new production with the factored prefix
            newProductions.push(`${prefix} ${newNT}`);

            // Create productions for the new non-terminal
            newNonTerminals[newNT]       =       orderProductions(suffixes,
allNonTerminals);
        }
        }

        return { newProductions, newNonTerminals };
}
```

lr.js

```
/**
 * Function to eliminate both direct and indirect left recursion from a context-
free grammar
 * @param {string} grammar - Grammar rules as a string
 * @return {string} - Grammar with left recursion eliminated
 */
function eliminateLeftRecursion(grammar) {
        // Parse the grammar into a structured format
        const parsedGrammar = parseGrammar(grammar);

        // Get all non-terminals in the grammar
        const nonTerminals = Object.keys(parsedGrammar);

        // Step 1: Arrange non-terminals in some order (A1, A2, ..., An)
        // We'll use the order they appear in the input

        // Step 2: Eliminate indirect left recursion
        for (let i = 0; i < nonTerminals.length; i++) {
        const Ai = nonTerminals[i];
```

```
        // For each previous non-terminal
        for (let j = 0; j < i; j++) {
                const Aj = nonTerminals[j];

                // Find productions of Ai that start with Aj
                const newProductions = [];
                const otherProductions = [];

                for (const prod of parsedGrammar[Ai]) {
                const parts = prod.split(' ').filter(p => p !== '');

                if (parts[0] === Aj) {
                        // This production starts with Aj
                        const beta = parts.slice(1).join(' ');

                        // Substitute each production of Aj
                        for (const AjProd of parsedGrammar[Aj]) {
                        if (AjProd === 'ϵ') {
                                // If Aj produces epsilon, just add beta
                                if (beta) {
                                newProductions.push(beta);
                                } else {
                                newProductions.push('ϵ');
                                }
                        } else {
                                // Otherwise combine Aj's production with beta
                                if (beta) {
                                newProductions.push(AjProd + ' ' + beta);
                                } else {
                                newProductions.push(AjProd);
                                }
                        }
                        }
                } else {
                        // Keep productions that don't start with Aj
                        otherProductions.push(prod);
                }
                }

                // Update the productions for Ai
                parsedGrammar[Ai] = [...otherProductions, ...newProductions];
        }

        // Step 3: Eliminate direct left recursion for Ai
        eliminateDirectLeftRecursion(parsedGrammar, Ai, nonTerminals);
        }

        // Order all productions in the final grammar
        for (const nonTerminal in parsedGrammar) {
        parsedGrammar[nonTerminal]                                              =
orderProductions(parsedGrammar[nonTerminal], nonTerminals);
        }

        // Convert the eliminated grammar back to string format
        return formatGrammar(parsedGrammar);
}
```

```
/**
 * Eliminate direct left recursion for a given non-terminal
 * @param {Object} grammar - The structured grammar object
 * @param {string} nonTerminal - The non-terminal to process
 * @param {Array} allNonTerminals - List of all non-terminals in the grammar
 */
function eliminateDirectLeftRecursion(grammar, nonTerminal, allNonTerminals) {
	const productions = grammar[nonTerminal];

	// Separate productions into those with left recursion and those without
	const recursiveProds = [];
	const nonRecursiveProds = [];

	for (const production of productions) {
	const parts = production.split(' ').filter(p => p !== '');

	if (parts[0] === nonTerminal) {
		recursiveProds.push(parts.slice(1).join(' ') || 'ϵ');
	} else {
		nonRecursiveProds.push(production);
	}
	}

	// Check if left recursion exists
	if (recursiveProds.length > 0) {
	// Create a new non-terminal
	const newNonTerminal = nonTerminal + "'";
	allNonTerminals.push(newNonTerminal);

	// Create new productions for the original non-terminal
	const newProductions = [];
	for (const prod of nonRecursiveProds) {
		if (prod === 'ϵ') {
		newProductions.push(newNonTerminal);
		} else {
		newProductions.push(prod + ' ' + newNonTerminal);
		}
	}
	grammar[nonTerminal]	=	orderProductions(newProductions,
allNonTerminals);

	// Create productions for the new non-terminal
	const newNTProductions = ['ϵ'];
	for (const prod of recursiveProds) {
		if (prod === 'ϵ') {
		newNTProductions.push(newNonTerminal);
		} else {
		newNTProductions.push(prod + ' ' + newNonTerminal);
		}
	}
	grammar[newNonTerminal]	=	orderProductions(newNTProductions,
allNonTerminals);
	}
}
```

tools.js

```
document.querySelectorAll(".btn-tool").forEach((button) => {
        button.addEventListener("click", () => {
        // console.log("Button clicked:", button.innerText);
        const buttonId = button.getAttribute("id");
        activeSection('sec-' + buttonId);
        // console.log("Button ID:", buttonId);
        activeButton(buttonId);
        });
}
);
function activeButton(buttonId) {
        const buttons = document.querySelectorAll(".btn-tool");
        buttons.forEach((button) => {
        if (button.getAttribute("id") === buttonId) {
                button.classList.add("active");
        } else {
                button.classList.remove("active");
        }
        });
}
function activeSection(sectionId) {
        document.getElementById(sectionId).classList.remove("hidden");
        document.querySelectorAll(".section").forEach((section) => {
        if (section.id !== sectionId) {
                section.classList.add("hidden");
        }
        });
}
document.getElementById("btn-process-lr").addEventListener("click", () => {
        const grammarInput = document.getElementById("lr-input").value;
        // console.log("Grammar Input:", grammarInput);
        const result = eliminateLeftRecursion(grammarInput);
        document.getElementById("lr-output").innerText = "";
        document.getElementById("lr-output").innerHTML = `<pre class="text-
black overflow-auto h-68">${result}</pre>`;
        // console.log("Left Recursion Eliminated:", result);
});
document.getElementById("btn-process-lf").addEventListener("click", () => {
        const grammarInput = document.getElementById("lf-input").value;
        // console.log("Grammar Input:", grammarInput);
        const result = eliminateLeftFactoring(grammarInput);
        document.getElementById("lf-output").innerText = "";
        document.getElementById("lf-output").innerHTML = `<pre class="text-
black overflow-auto h-68">${result}</pre>`;
        // console.log("Left Factoring Eliminated:", result);
}
);
document.getElementById("btn-process-ff").addEventListener("click", () => {
        const grammarInput = document.getElementById("ff-input").value;
        // console.log("Grammar Input:", grammarInput);
        const result = computeFirstAndFollowSets(grammarInput);
        console.log("First and Follow Sets Computed:", result);
        document.getElementById("ff-output").innerText = "";
        let tableHTML = "<table class='text-sm md:text-lg text-black overflow-
auto w-full rounded-box border-collapse border border-gray-300'><thead><tr><th
```

```
class='border border-gray-300 px-4 py-2'>Non-Terminal</th><th class='border
border-gray-300 px-4 py-2'>First</th><th class='border border-gray-300 px-4 py-
2'>Follow</th></tr></thead><tbody>";


        for (const nonTerminal of Object.keys(result.first)) {
        const firstSet = result.first[nonTerminal].join(" ");  // No quotes
        const followSet = result.follow[nonTerminal].join(" ");  // No quotes

        tableHTML += `<tr>
               <td class='border border-gray-300 px-4 py-2'>${nonTerminal}</td>
                <td class='border border-gray-300 px-4 py-2'>${firstSet}</td>
                <td class='border border-gray-300 px-4 py-2'>${followSet}</td>
        </tr>`;
        }

        tableHTML += "</tbody></table>";
        document.getElementById("ff-output").innerHTML = tableHTML;


});
```

= The End =

"MASTERING COMPILERS IS NOT JUST ABOUT PARSING CODE
—IT'S ABOUT PARSING COMPLEXITY INTO CLARITY."

Thank you for picking up this book!

Compiler Design can seem intimidating, but with the right guidance, it becomes a fascinating journey. I've written this book with clarity and practice in mind—just like I'd teach it in class. I hope it helps you learn, build, and enjoy the process.

Warm regards,
Sajjadul Islam Somon
CSE, Daffodil International University
Daffodil Smart City, Birulia, Savar, Dhaka, Bangladesh