Compiler Design

Dr. Soumen Das

Compiler-Introduction

- A **compiler** is software that translates or converts a program written in a high-level language (Source Language) into a low-level language (Machine Language).
- Compiler design is the process of developing a program or software that converts human-written code into machine code.
- It involves many stages like lexical analysis, parsing, semantic analysis, code generation, optimization, etc.
- The Key objective of compiler design is to automate the translation process, the correctness of output, and reporting errors in source code.

The compiler is used by programming languages such as C, C++, C#, Java,

Compiler-Introduction

Why Do We Learn Compiler Design?

A computer is a logical assembly of software and hardware. The hardware understands a language that is hard for humans to understand. So, we write programs in a high-level language, that is less complex for humans to understand and maintain in thoughts. Now, a series of transformations have been applied to high-level language programs to convert them into machine language.

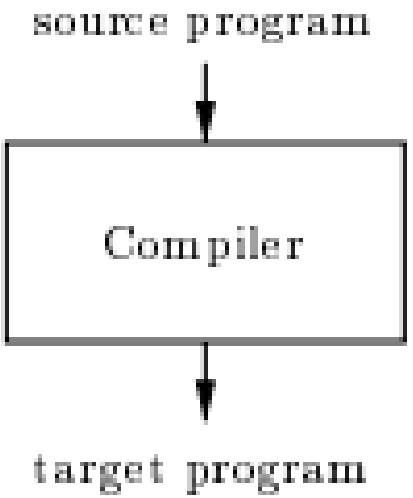
Compiler-Introduction

Primary task of compiler

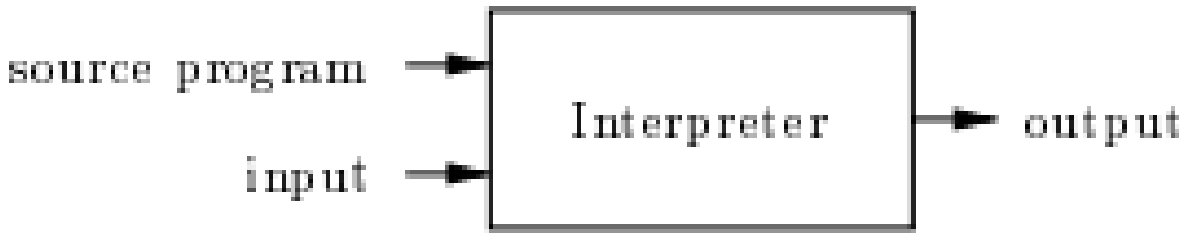
- Generate target program in the execution domain keeping in mind the architectural details of the computer where it is implemented. While translating it must preserve the meaning of specification across translation from source to target language.
- Possess diagnostic features to recognize any violation in program specification at syntactic as well as semantic levels.
- Capable of generating compact and efficient target code and optimize the use of system resources.

To achieve these task the compiler goes through several phases and passes

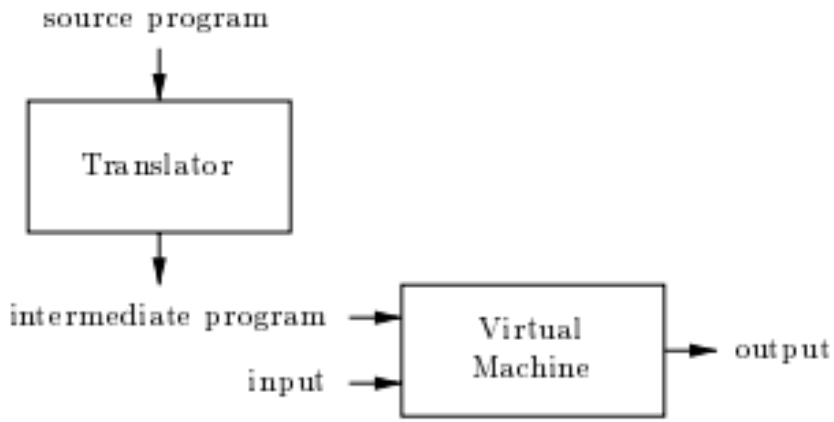
Compiler-Introduction

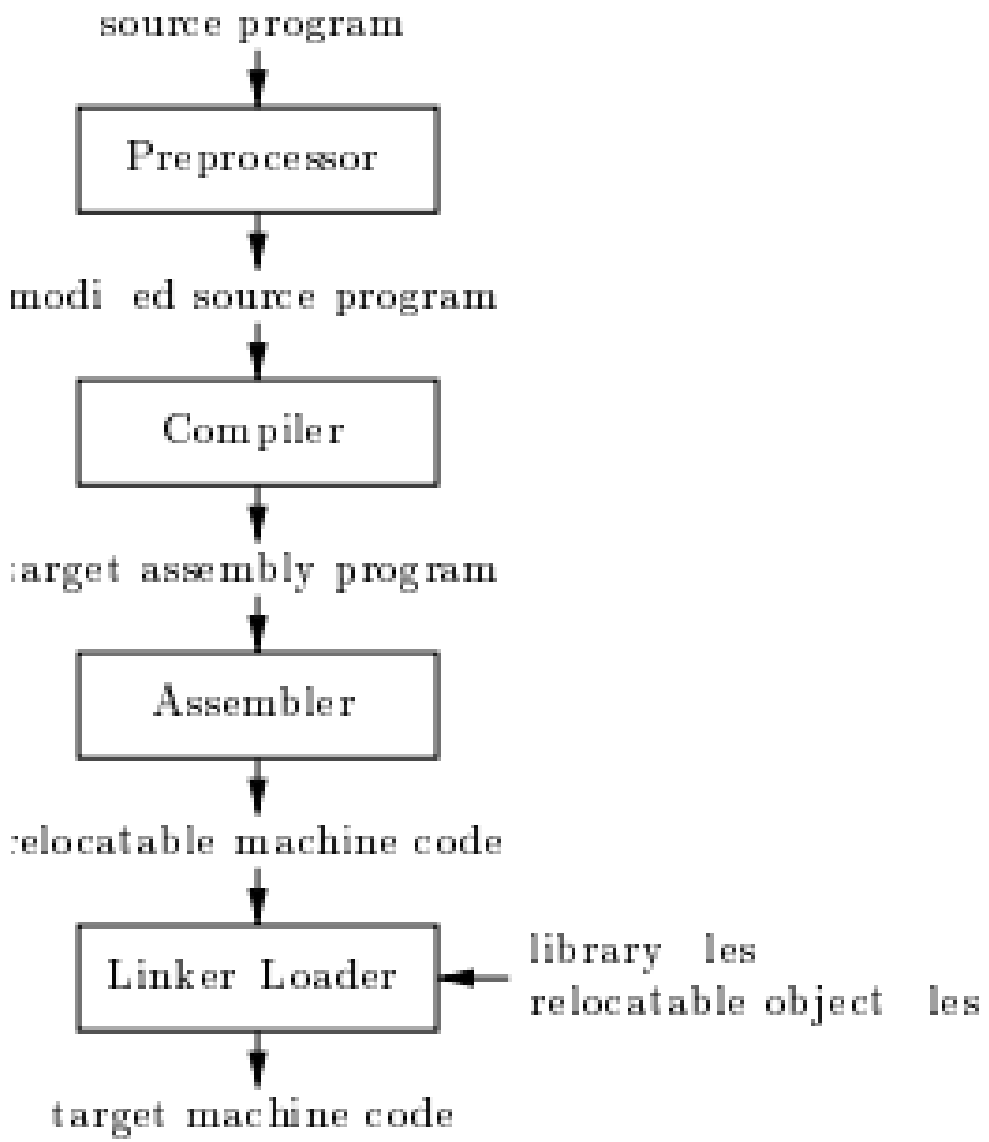


An interpreter is another common kind of language processor. Instead of producing a target program as a translation, an interpreter appears to directly execute the operations in the source program on inputs supplied by the user, as shown



A hybrid compiler




source program
Preprocessor
modi ed source program
Compiler
:arget assembly program
Assembler
:elocatable machine code
Linker Loader
library les
relocatable object les
target machine code

Phases and Passes

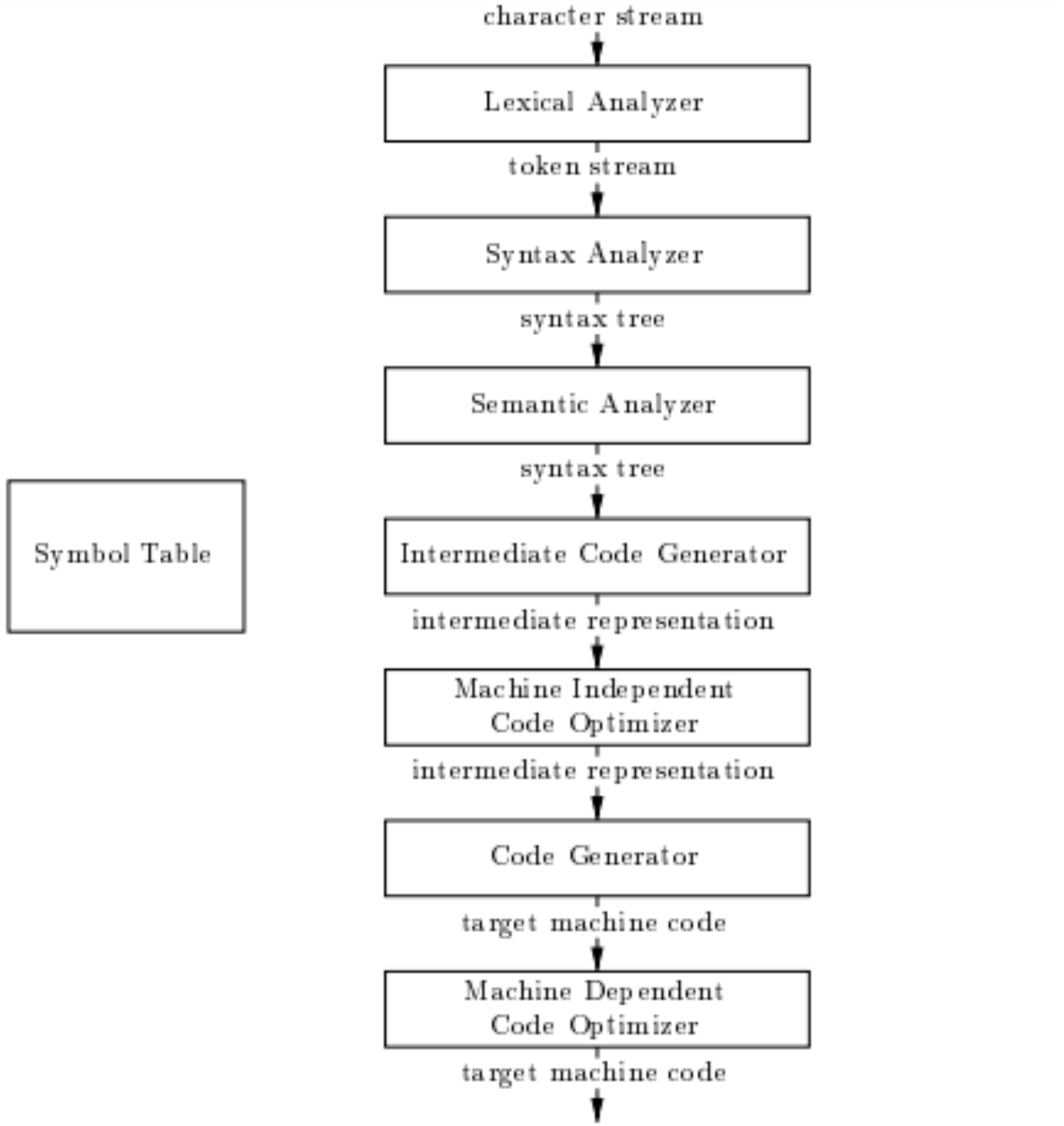
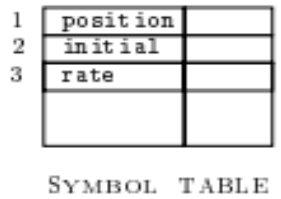
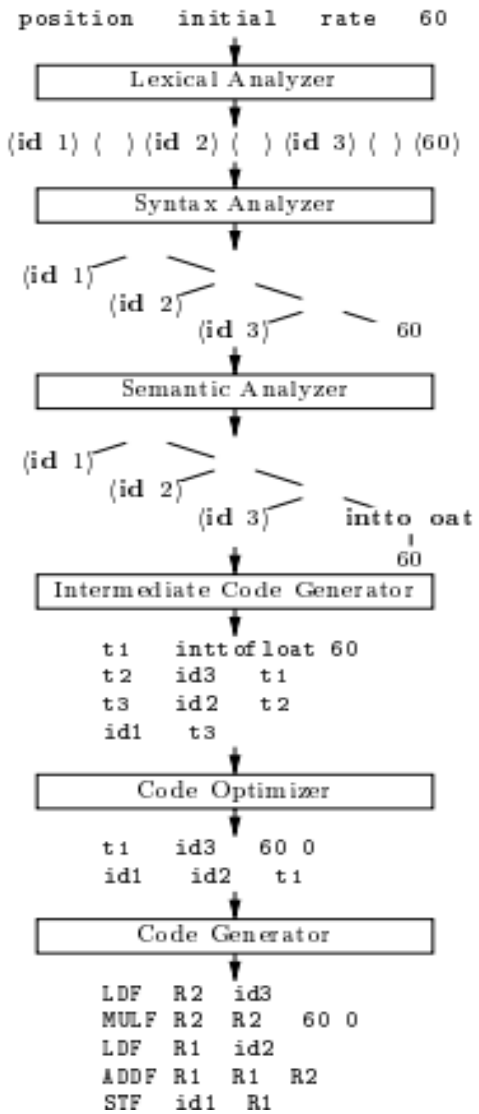


Figure 1 6 Phases of a compiler

Translation of an assignment statement





Passes in Compiler:

- A pass is a component where parts of one or more phases of the compiler are combined when a compiler is implemented. A pass reads or scans the instructions of the source program, or the output produced by the previous pass, which makes necessary transformation specified by its phases.
- There are generally two types of passes

1. Single-pass

2. Multi-pass

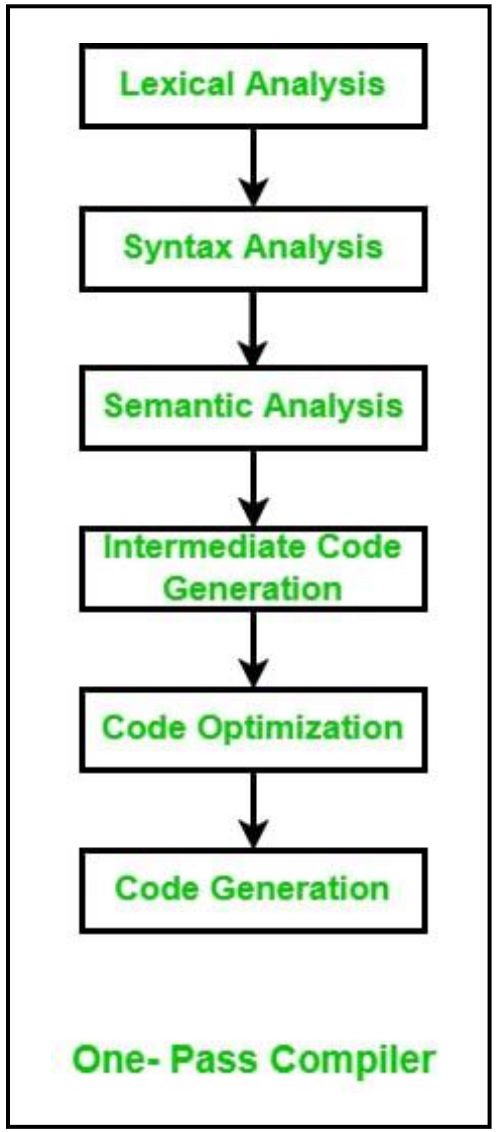
Cont..

1. Single-Pass – In One-pass all the phases are grouped into one phase. The six phases are included here in one pass.

2. Multi-Pass – In Two-pass the phases are divided into two parts i.e. Analysis or Front End part of the compiler and the synthesis part or back end part of the compiler.

Single-Pass Compiler

- Single pass compiler almost never done, early **Pascal compiler** did this as an introduction.



One-Pass

- A one-pass/single-pass compiler is a type of compiler that passes through the part of each compilation unit exactly once.
- Single pass compiler is faster and smaller than the multi-pass compiler.
- A disadvantage of a single-pass compiler is that it is less efficient in comparison with the multipass compiler.
- A single pass compiler is one that processes the input *exactly once* , so going directly from lexical analysis to code generator, and then going back for the next read.

Single-Pass

Problems with Single Pass Compiler

- We can not optimize very well due to the context of expressions are limited.
- As we can’t back up and process, it again so grammar should be limited or simplified.
- Command interpreters such as *bash/sh/tcsh* can be considered Single pass compilers, but they also execute entries as soon as they are processed.

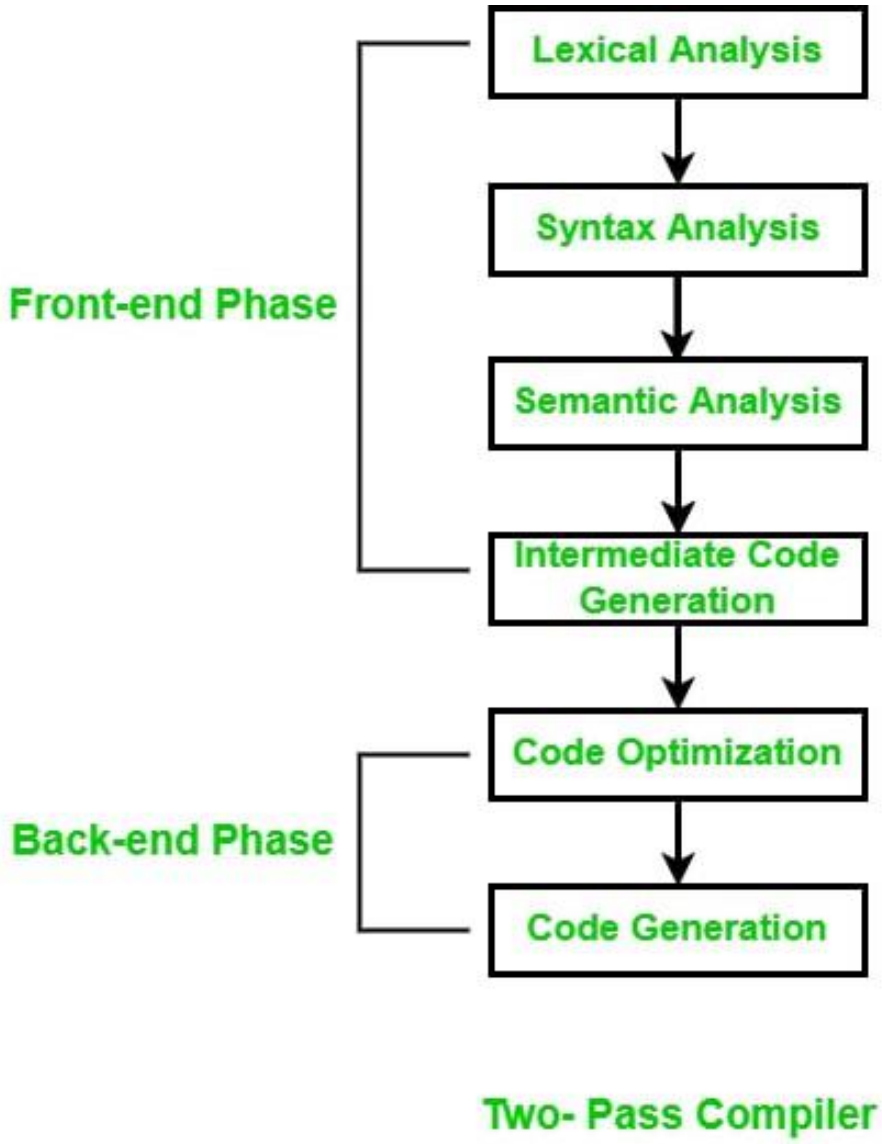
Two-Pass

First Pass is referred as Front end

- Analytic part
- Platform independent

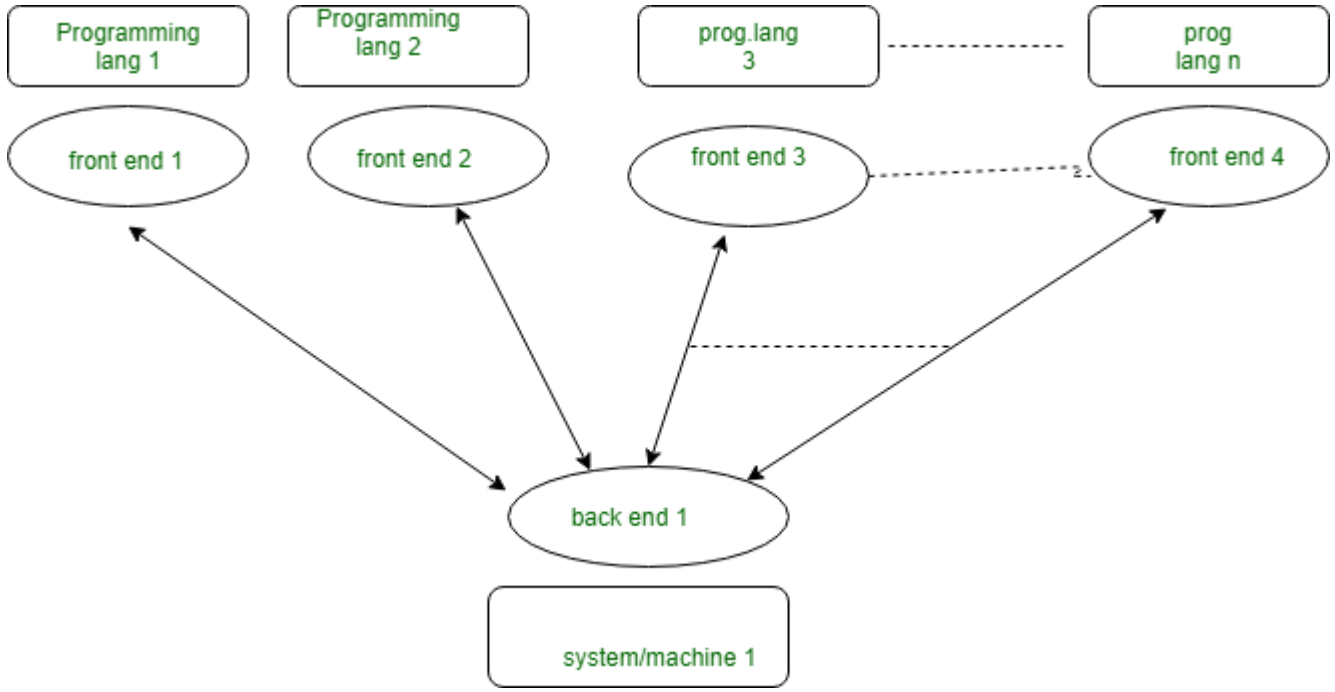
Second Pass is referred as Back end

- Synthesis Part
- Platform Dependent



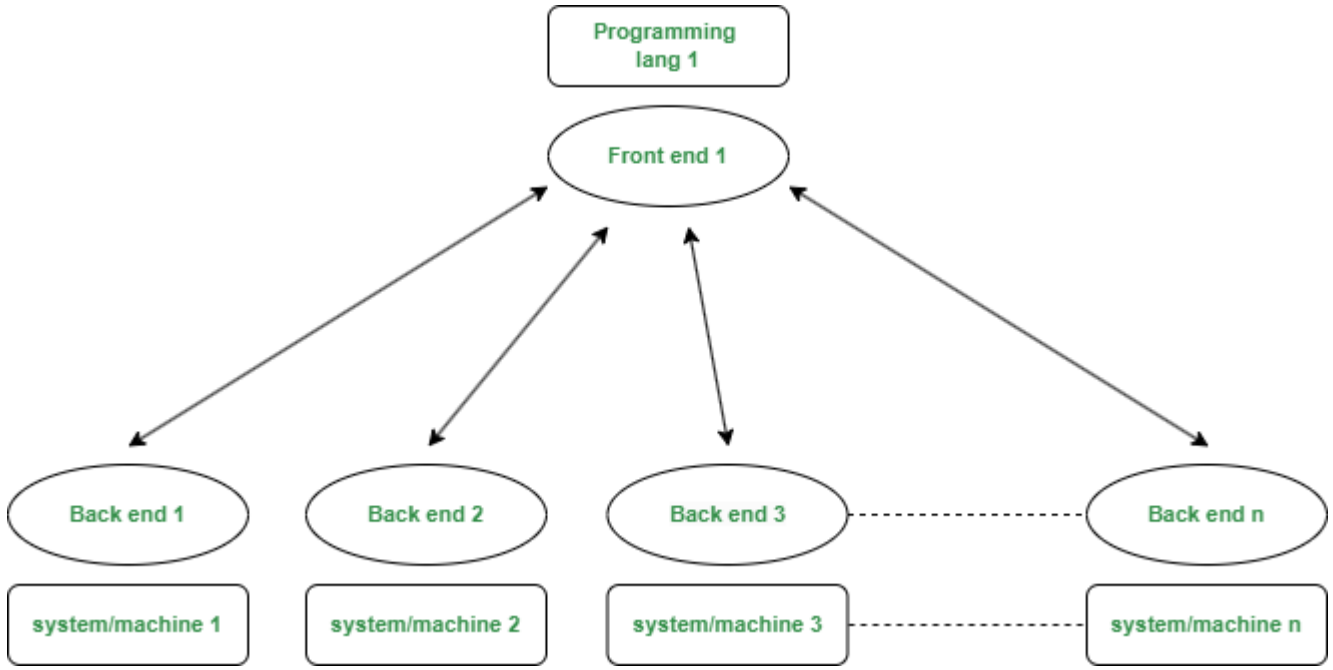
Problems that can be Solved With Multi-Pass Compiler

First: If we want to design a compiler for a different programming language for the same machine. In this case for each programming language, there is a requirement to make the Front end/first pass for each of them and only one Back end/second pass as:



Problems that can be Solved With Multi-Pass Compiler

Second: If we want to design a compiler for the same programming language for different machines/systems. In this case, we make different Back end for different Machine/system and make only one **Front end** for the same programming language as:



Problems that can be Solved With Multi-Pass Compiler

One Pass Compiler	Two Pass Compiler
It performs Translation in one pass	It performs Translation in two pass
It scans the entire file only once.	It requires two passes to scan the source file.
It generates Intermediate code	It does not generate Intermediate code
It is faster than two pass assembler	It is slower than two pass assembler
A loader is not required	A loader is required.
No object program is written.	A loader is required as the object code is generated.

Example: C and Pascal uses One Pass Compiler.

Example: Modula-2 uses Multi Pass Compiler.

Bootstrapping

- **Bootstrapping** is a process in which simple language is used to translate more complicated program which in turn may handle for more complicated program.

- It is an approach for making a self-compiling compiler that is a compiler written in the source programming language that it determine to compile.

Ex: A bootstrap compiler can compile the compiler and thus you can use this compiled compiler to compile everything else and the future versions of itself.

- Bootstrapping in compiler design refers to the process of developing a compiler using an existing compiler for the same or a different programming language.

- Bootstrapping is used to produce a self-hosting compiler. Self-hosting compiler is a type of compiler that can compile its own source code.

Use of Bootstrapping

- It can allow new programming languages and compilers to be developed starting from actual ones.
- It allows new features to be combined with a programming language and its compiler.
- It also allows new optimizations to be added to compilers.
- It allows languages and compilers to be transferred between processors with different instruction sets

Advantage of Bootstrapping

- Compiler development can be performed in the higher-level language being compiled.
- It is a non-trivial test of the language being compiled.
- It is an inclusive consistency check as it must be capable of recreating its object code.

Bootstrapping

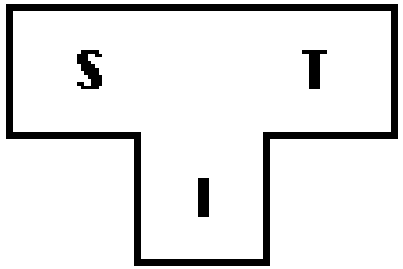
For bootstrapping a compiler can be characterized by three languages

1.Source Language

2.Target Language

3.Implementation Language

The T- diagram shows a compiler SCIT for Source S, Target T, implemented in I.

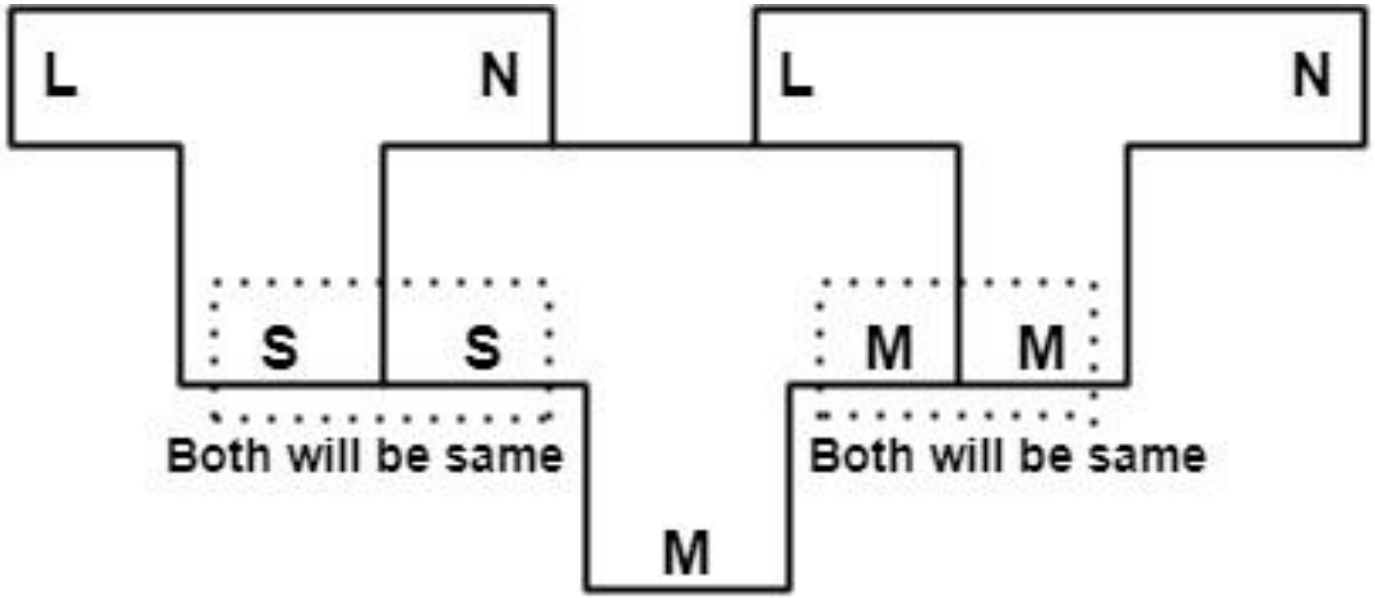


Cross Compiler

A compiler is characterized by three languages as its source language, its object language, and the language in which it is written. These languages may be quite different. A compiler can run on one machine and produce target code for another machine. Such a compiler is known as a cross-compiler.

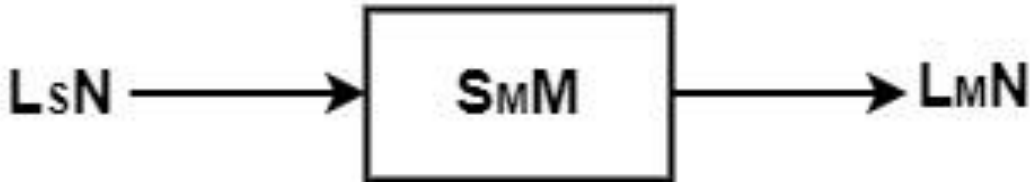
- If a current compiler for S runs on machine M and generates a program for M, it is defined by SMM.
- If LSN runs through SMM, we get a compiler LMN, i.e., a compiler from L to N that runs on M.

Cross Compiler



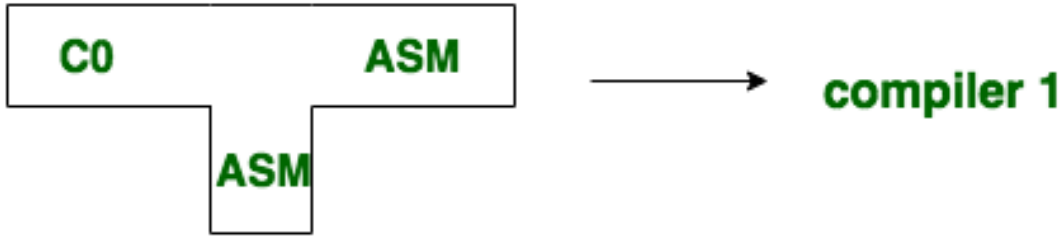
i.e., $L_SN + S_MM = L_MN$

It can also be represented as

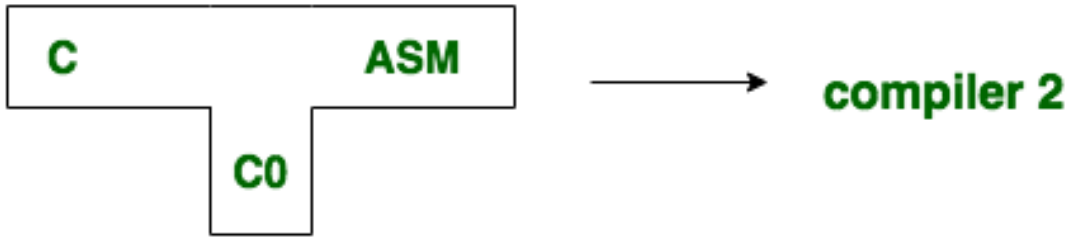


Cont..

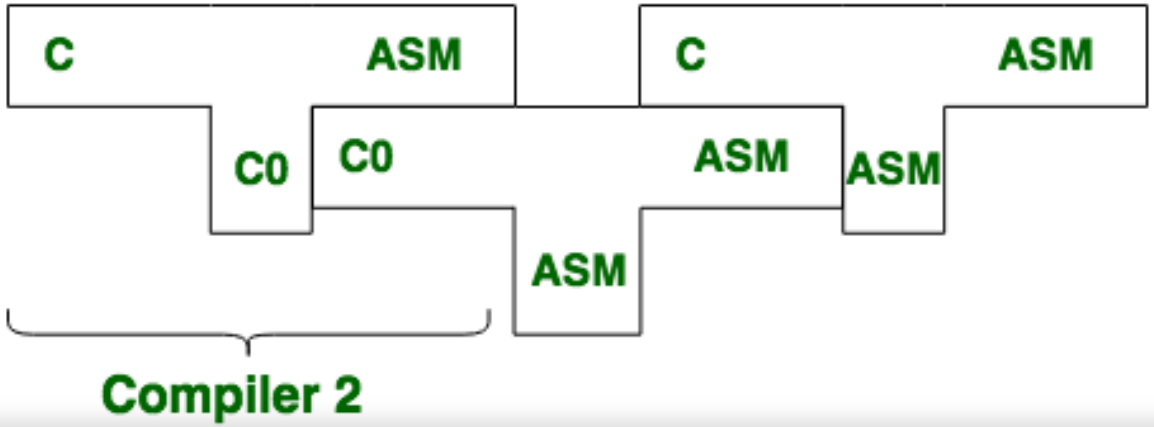
- **Step-1:** First we write a compiler for a small of C in assembly language.



- **Step-2:** Then using with small subset of C i.e. C0, for the source language c the compiler is written.



- **Step-3:** Finally we compile the second compiler. using compiler 1 the compiler 2 is compiled.



Cont..

- **Step-4:** Thus we get a compiler written in ASM which compiles C and generates code in ASM.

Advantages:

1. Bootstrapping ensures that the compiler is compatible with the language it is designed to compile, as it is written in the same language.

2. It allows for greater control over the optimization and code generation process.

3. It provides a high level of confidence in the correctness of the compiler because it is self-hosted.

Disadvantages:

1. It can be a time-consuming process, especially for complex languages or compilers.

2. Debugging a bootstrapped compiler can be challenging since any errors or bugs in the compiler will affect the subsequent versions of the compiler.

3. Bootstrapping requires that a minimal version of the compiler be written in a different language, which can introduce compatibility issues between the two languages.

4. Overall, bootstrapping is a useful technique in compiler design, but it requires careful planning and execution to ensure that the benefits outweigh the drawbacks.

•

Regular Expression

- The language accepted by finite automata can be easily described by simple expressions called Regular Expressions. It is the most effective way to represent any language.
- The languages accepted by some regular expression are referred to as Regular languages.
- A regular expression can also be described as a sequence of pattern that defines a string.
- Regular expressions are used to match character combinations in strings. String searching algorithm used this pattern to find the operations on a string.

For instance:

- In a regular expression, x* means zero or more occurrence of x. It can generate {e, x, xx, xxx, xxxx, ...}
- In a regular expression, x^+ means one or more occurrence of x. It can generate {x, xx, xxx, xxxx, ...}

Regular Language

- A regular language is a type of formal language that can be expressed using regular expressions and can be recognized by finite automata. Regular languages are the simplest class in the Chomsky hierarchy of languages.

Characteristics of Regular Languages

1. **Closure Properties:** Regular languages are closed under various operations:
   Union: If L1 and L2 are regular languages, then L1 ∪ L2 is also regular.
   Concatenation: If L1 and L2 are regular languages, then L1 · L2 is also regular.
   Kleene Star: If L is a regular language, then L* (zero or more concatenations of L) is also regular.
   Intersection: If L1 and L2 are regular languages, then L1 ∩ L2 is also regular.
   Complement: If L is a regular language, then its complement is also regular.

2. **Recognizability:** Regular languages can be recognized by:
   Finite Automata (FA): A deterministic finite automaton (DFA) or nondeterministic finite automaton (NFA) can recognize regular languages.
   Regular Expressions: Any language that can be described by a regular expression is a regular language.

3. Decidability: Problems such as emptiness, finiteness, membership, and equivalence for regular languages are decidable, meaning there exists an algorithm to solve these problems.

Regular Language

Examples of Regular Languages

1. **Binary Strings with Even Number of Zeros:**
   - Regular expression: (1*01*01*)*

2. **Strings Over {a, b} Ending in "ab":**
   - Regular expression: (a + b)*ab

3. All Strings Over {0, 1}:
   - Regular expression: (0 + 1)*

Non-Regular Languages
Certain languages are not regular and require more complex computational models like context-free grammars or Turing machines. For example:
- The language L = {a^n b^n | n ≥ 0} is not regular because it requires counting, which finite automata cannot handle.

Finite Automata

- We shall now discover how Lex turns its input program into a lexical analyzer.
- At the heart of the transition is the formalism known as finite automata.
- These are essentially graphs, like transition diagrams, with a few differences:

1. Finite automata are recognizers; they simply say "yes" or "no" about each possible input string.
2. Finite automata come in two flavors:

(a) Nondeterministic finite automata (NFA) have no restrictions on the labels of their edges. A symbol can label several edges out of the same state, and E, the empty string, is a possible label.

(b) Deterministic finite automata (DFA) have, for each state, and for each symbol of its input alphabet exactly one edge with that symbol leaving that state.

Nondeterministic Finite Automata

- Both deterministic and nondeterministic finite automata are capable of recognizing the same languages.

- These languages are exactly the same languages, called the regular languages, that regular expressions can described. NFA consist of

1. A finite set of states S.
2. A set of input symbols Σ, the *input alphabet*. We assume that ϵ, which stands for the empty string, is never a member of Σ.
3. A *transition function* that gives, for each state, and for each symbol in $\Sigma \cup \{\epsilon\}$ a set of *next states*.
4. A state s_0 from S that is distinguished as the *start state* (or *initial state*).
5. A set of states F, a subset of S, that is distinguished as the *accepting states* (or *final states*).

We can represent either an NFA or DFA by a *transition graph*, where the nodes are states and the labeled edges represent the transition function. There is an edge labeled a from state s to state t if and only if t is one of the next states for state s and input a. This graph is very much like a transition diagram, except:

Nondeterministic Finite Automata

4. A state S_0 from S that is distinguished as the start state (or initial state).

5. A set of states F, a subset of S, that is distinguished as the accepting states (or final states).

An NFA is defined by a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where:

- Q is a finite set of states.
- Σ is a finite set of input symbols (alphabet).
- δ is the transition function $\delta : Q \times (\Sigma \cup \{\epsilon\}) \to 2^Q$, mapping a state and an input symbol to a set of possible next states.
- $q_0 \in Q$ is the initial state.
- $F \subseteq Q$ is the set of accepting (final) states.

Nondeterministic Finite Automata

a) The same symbol can label edges from one state to several different states, and

b) An edge may be labeled by ϵ, the empty string, instead of, or in addition to, symbols from the input alphabet.

Example 3.14: The transition graph for an NFA recognizing the language of regular expression **(a|b)*abb** is shown in Fig. 3.24. This abstract example, describing all strings of *a*'s and *b*'s ending in the particular string *abb*, will be used throughout this section. It is similar to regular expressions that describe languages of real interest, however. For instance, an expression describing all files whose name ends in `.o` is **any*.o**, where **any** stands for any printable character.

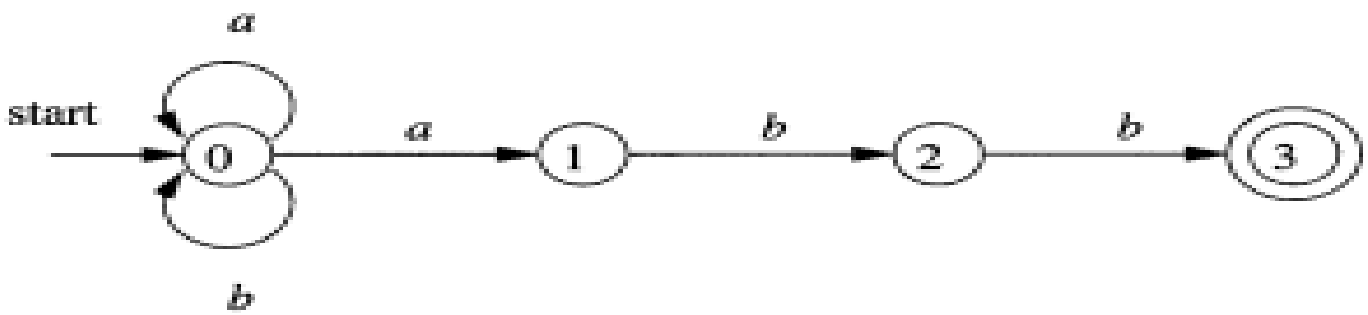


Figure 3.24: A nondeterministic finite automaton

Nondeterministic Finite Automata

We can also represent an NFA by a transition table, whose rows correspond to states, and whose columns correspond to the input symbols and €. The entry for a given state and input is the value of the transition function applied to those arguments. If the transition function has no information about that state-input pair, we put ∅ in the table for the pair.

STATE	a	b	ϵ
0	$\{0, 1\}$	$\{0\}$	$\emptyset$
1	$\emptyset$	$\{2\}$	$\emptyset$
2	$\emptyset$	$\{3\}$	$\emptyset$
3	$\emptyset$	$\emptyset$	$\emptyset$

Nondeterministic Finite Automata

An NFA accepts input string x if and only if there is some path in the transition graph from the start state to one of the accepting states, such that the symbols along the path spell out x. Note that c labels along the path are effectively ignored, since the empty string does not contribute to the string constructed along the path.

Example 3.16: The string *aabb* is accepted by the NFA of Fig. 3.24. The path labeled by *aabb* from state 0 to state 3 demonstrating this fact is:

$$0 \xrightarrow{a} 0 \xrightarrow{a} 1 \xrightarrow{b} 2 \xrightarrow{b} 3$$

Note that several paths labeled by the same string may lead to different states. For instance, path

$$0 \xrightarrow{a} 0 \xrightarrow{a} 0 \xrightarrow{b} 0 \xrightarrow{b} 0$$

is another path from state 0 labeled by the string *aabb*. This path leads to state 0, which is not accepting. However, remember that an NFA accepts a string as long as *some* path labeled by that string leads from the start state to an accepting state. The existence of other paths leading to a nonaccepting state is irrelevant. □

The *language defined* (or *accepted*) by an NFA is the set of strings labeling some path from the start to an accepting state. As was mentioned, the NFA of Fig. 3.24 defines the same language as does the regular expression (a|b)*abb, that is, all strings from the alphabet $\{a, b\}$ that end in *abb*. We may use $L(A)$ to stand for the language accepted by automaton A.

Deterministic Finite Automata

A deterministic finite automaton (DFA) is a special case of an NFA where:

- There are no moves on input E, and 2.
- For each state s and input symbol a, there is exactly one edge out of s labeled a.

Exercise

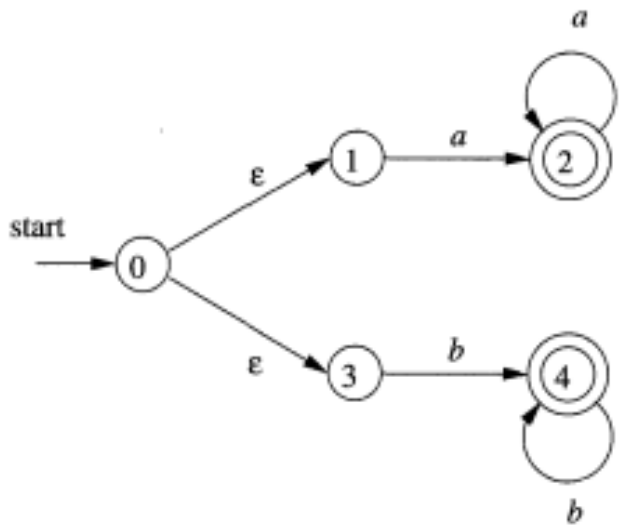


Figure 3.26: NFA accepting **aa*|bb***

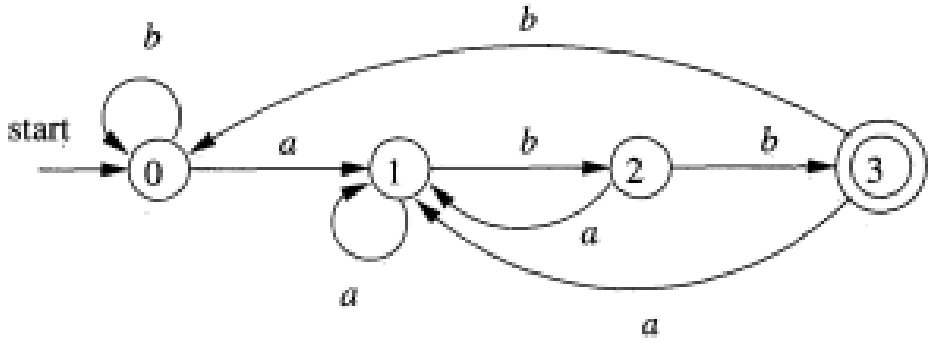


Figure 3.28: DFA accepting **(a|b)*abb**

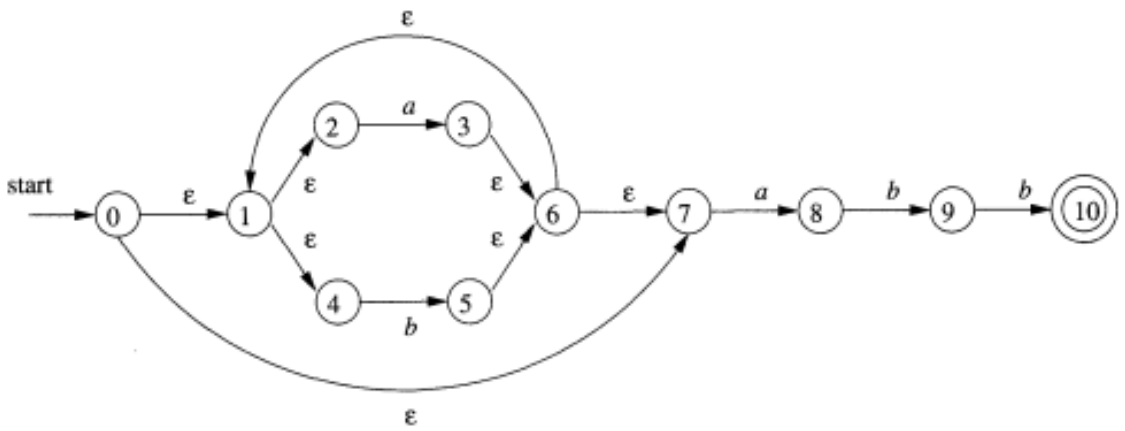


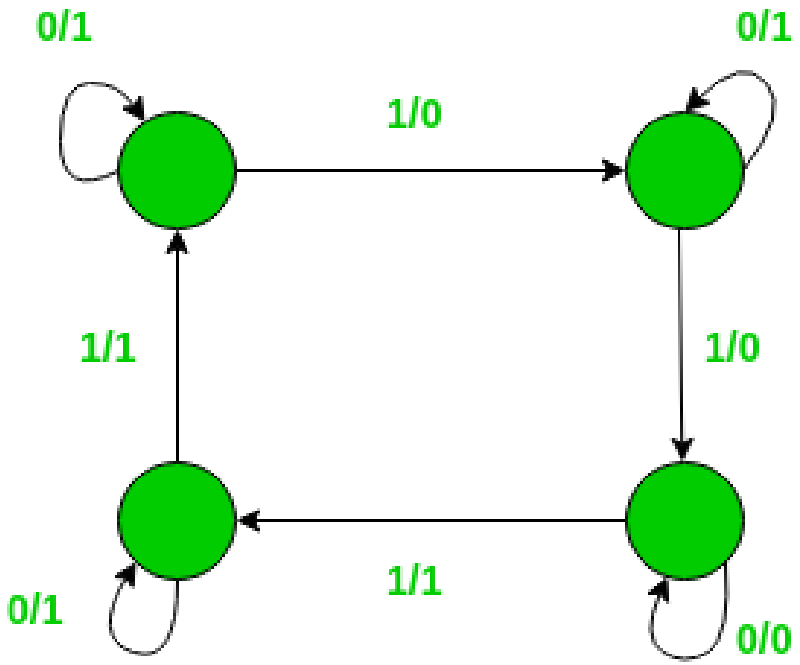
Figure 3.34: NFA N for **(a|b)*abb**

Mealy Machine

What is Mealy Machine?

What is Mealy Machine?

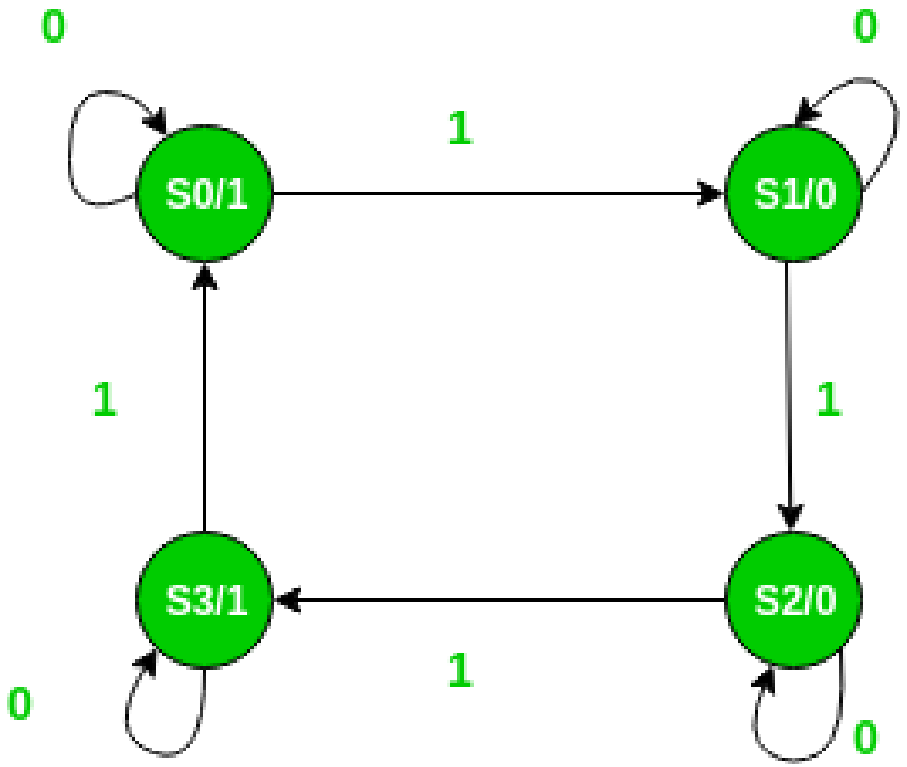
Mealy Machine is defined as a machine in the theory of computation whose output values are determined by both its current state and current inputs. In this machine at most one transition is possible.



Moore Machine

Moore Machine

Moore's machine is defined as a machine in the theory of computation whose output values are determined only by its current state. It has also 6 tuples



Application of Regular Expression in Lexical Analysis

- Regular expressions (regex) are essential in lexical analysis, the first phase of a compiler. Lexical analysis converts source code into tokens, which are defined by regex patterns.

Token Specification

Regular expressions define various token types such as keywords, identifiers, literals, and operators. Example regex patterns:

- **Keywords:** \b(if|while|for|return)\b
- **Identifiers:** [a-zA-Z_][a-zA-Z0-9_]*
- **Operators:** \+|\-|*|\/

Application of Regular Expression in Lexical Analysis

Tokenization Process:

Lexical Analyzer (Lexer): The lexer uses regular expressions to scan the source code and match sequences of characters against the predefined regex patterns.

Token Matching: When a match is found, the corresponding token type is created, and the lexer moves to the next part of the source code.

Error Handling: If no match is found for a sequence of characters, the lexer may raise an error, indicating a lexical error in the source code.

Application of Regular Expression in Lexical Analysis

Use in Compiler Tools

Lex/Yacc and Flex/Bison: Tools like Lex (or Flex) and Yacc (or Bison) leverage regular expressions to automate the generation of lexical analyzers. In these tools, developers specify token patterns using regular expressions, and the tool generates the code for the lexer.

Application of Regular Expression in Lexical Analysis

Efficiency and Optimization:

Finite Automata: Regular expressions are often compiled into finite automata (either deterministic or non-deterministic) to improve the efficiency of the tokenization process.

Lookahead and Backtracking: Some lexical analyzers use lookahead or backtracking mechanisms to resolve ambiguities and ensure the longest match for tokens.

Regular Expressions and their Applications to Lexical Analysis,

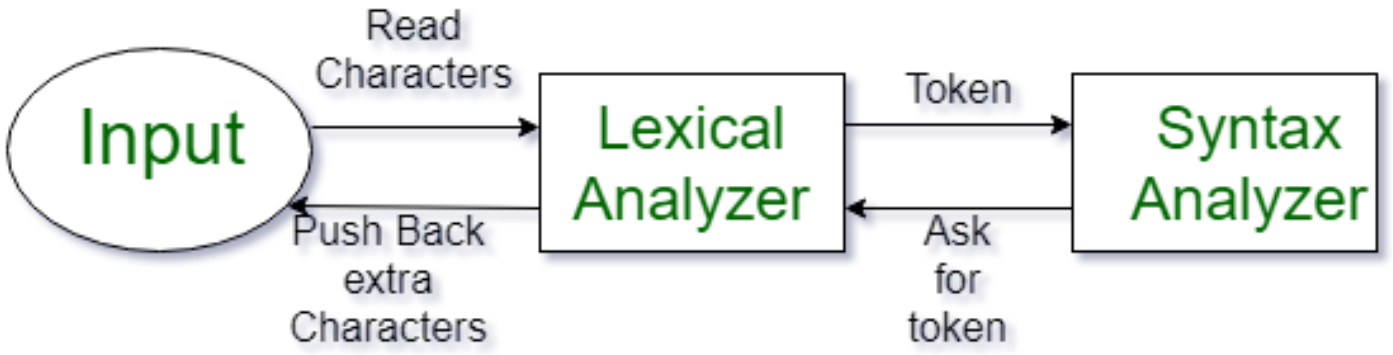
- Lexical Analysis is the first phase of a compiler that takes the input as a source code written in a high-level language.

- The purpose of lexical analysis is that it aims to read the input code and break it down into meaningful elements called **tokens**.

Cont..

What is a Token?

- A lexical token is a sequence of characters that can be treated as a unit in the grammar of the programming languages.



Example of tokens

- Type token (id, number, real, . . .)
- Punctuation tokens (IF, void, return, . . .)
- Alphabetic tokens (keywords)

Keywords; Examples - for, while, if etc.
Identifier; Examples - Variable name, function name, etc.
Operators; Examples '+', '++', '-' etc.
Separators; Examples ',' ';' etc

Example of Non-Tokens

- Comments, preprocessor directive, macros, blanks, tabs, newline, etc.

What is a Lexeme?

- The sequence of characters in source program that matched by a pattern to form the corresponding token or a sequence of input characters that comprises a single token is called a lexeme.

eg- “float”, “abs_zero_Kelvin”, “=”, “-”, “273”, “;” .

Implementation of Lexical Analyzers and Syntax Analysis
Introduction to Lexical Analyzers

- - Converts source code into tokens.
- - Removes white spaces and comments.
- - Identifies keywords, identifiers, operators, and literals.

- Example:
- `int x = 10;` → Tokens: `int`, `x`, `=`, `10`, `;`

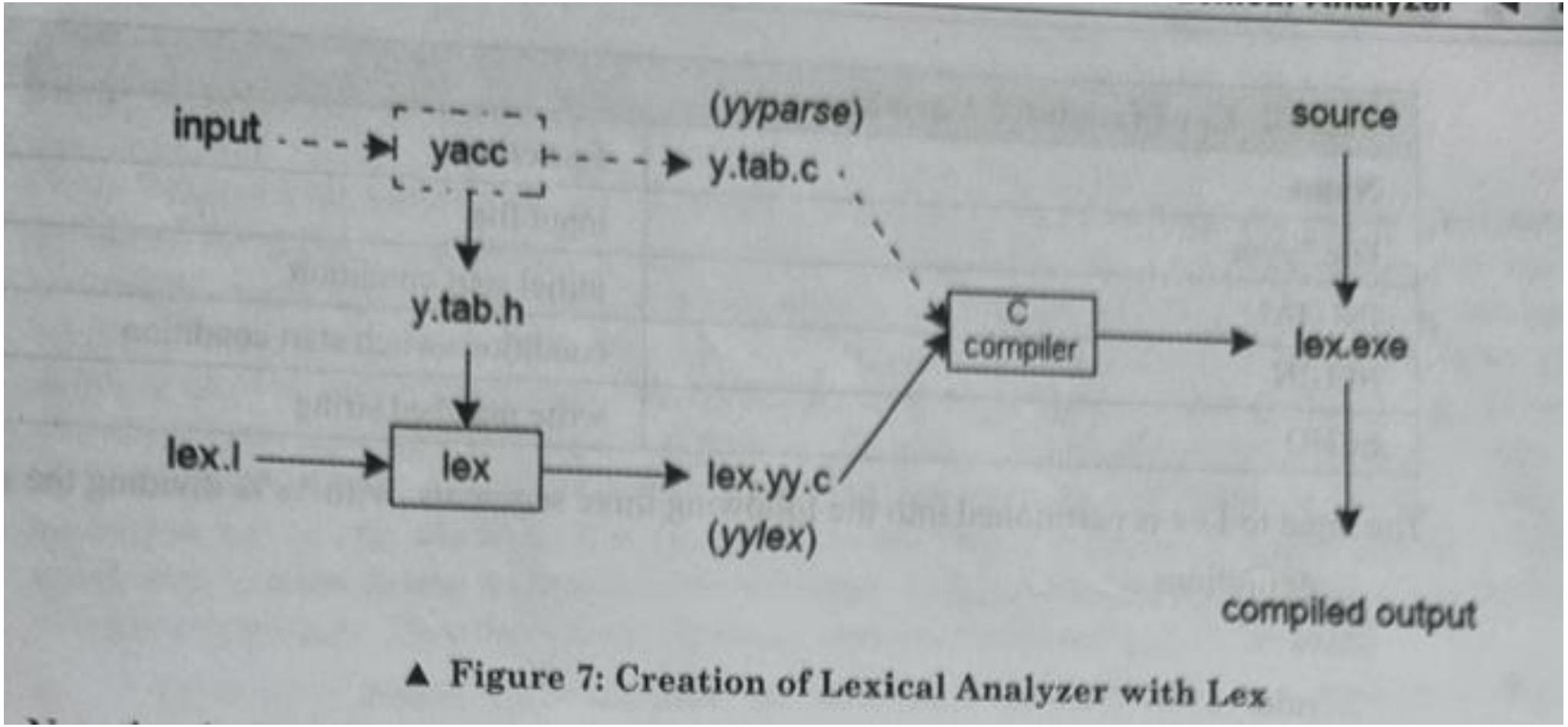
Example

```
int main() {
    int x = 42;
    if (x > 0) {
        return x;
    }
}
```

Example

- int → Keyword
- main → Identifier
- () → Delimiters
- {} → Delimiters
- 42 → Integer literal
- = → Assignment operator
- > → Relational operator
- return → Keyword

Implementation of Lexical Analyzer



▲ Figure 7: Creation of Lexical Analyzer with Lex

Pattern Matching Primitives

Metacharacter	Matches
.	Any character except newline
\n	Newline
*	Zero or more copies of the expression
+	One or more copies of the expression
?	Zero or one copy of the expression
^	Beginning of line
$	End of line
x|y	x or y
(xy)*	One or more copies of xy (concatenation)
"x+y"	Literal "x+y"
[]	Represents a character class

Pattern Matching Primitives

- The metacharacters are used by regular expressions in the Lex. Note that when two patterns match for some string, the longest match is considered, and when more than one match are of the same
- length, the first pattern listed is used.
- The predefined variables and their functions that are used by Lex:

Lex Predefined Variables

Name	Function
int yylex (void)	Call to invoke lexer, returns token
char *yytext	Pointer to matched string
yyleng	Length of matched string
yylval	Value associated with token
int yywrap (void)	Wrapup, return 1 if done, 0 if not done
File *yyout	Output file

Lex Predefined Variables

Name	Function
File *yyin	input file
INITIAL	initial start condition
BEGIN	condition switch start condition
ECHO	write matched string

Example of Lex program

```
digit [0-9]
letter [A-Za-z]
%{
int count;
%}
%%
/* match the identifier */
{letter}({letter}|{digit})* count++;
%%
int main(void) {
yylex();
printf("number of identifiers = %d\n", count);
return 0;
}

The following specification is a simple example of a  program that does
nothing at all.
%%
.
\n
```

Problems of Lex program

Ambiguous or Conflicting Patterns

Problem: When two or more patterns match the same input, Lex chooses the longest match. If multiple matches have the same length, the first rule listed in the Lex file is used.

Solution

- Carefully order patterns in your Lex specification file. Ensure more specific patterns appear before generic ones.
- Use BEGIN states to separate contexts, reducing ambiguity.

Problems of Lex program

Handling Input Streams

Problem: The yyin input stream might not handle complex or multi-file input well.

Solution

- If processing multiple files, close and reopen

Problems of Lex program

Buffer Size Limitations Problem:

Large input files may exceed the default buffer size, causing errors.

Solution:

Define a larger buffer size in your Lex file

Problems of Lex program

Complex Lexical Rules Problem:

Some token rules can be difficult to express in regular expressions, or Lex rules can grow too complicated to manage.

Solution:

- Break down complex rules into simpler components. Use helper functions for post-processing complex patterns.

Problems of Lex program

Poor Integration with C Code Problem:
Mixing Lex-generated code with other parts of your program can lead to undefined behavior or poor readability.

Solution:

- Encapsulate Lex-generated code using proper modularization techniques.

- Use yywrap() effectively to control what happens after Lex finishes processing input.

Problems of Lex program

Limited Documentation for Advanced Features Problem:

Advanced features of Lex may not be well-documented.

Solution:

- Look for additional resources, such as books or online tutorials, to deepen your understanding.
- Use open-source alternatives like Flex, which is better documented and widely used.

Formal Grammars and Their Applications to Syntax Analysis

Formal grammar

- Formal grammar is a set of rules. It is used to identify correct or incorrect strings of tokens in a language. The formal grammar is represented as G.
- Formal grammar is used to generate all possible strings over the alphabet that is syntactically correct in the language.
- Formal grammar is used mostly in the syntactic analysis phase (parsing) particularly during the compilation.
- Formal grammar G is written as follows:

G = <V, N, P, S>

Formal Grammars and Their Applications to Syntax Analysis

Where:

N describes a finite set of non-terminal symbols.
V describes a finite set of terminal symbols.
P describes a set of production rules
S is the start symbol.

L = {a, b}, N = {S, R, B}

S = bR

R = aR

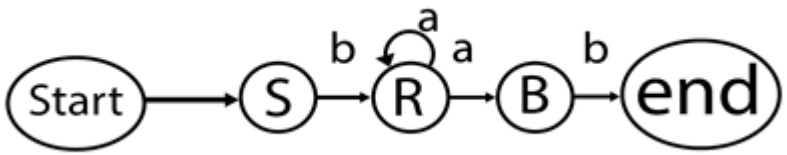
R = aB

B = b

Through this production we can produce some strings like: bab, baab, baaab etc.

Formal Grammars and Their Applications to Syntax Analysis

his production describes the string of shape ba^nab.



Formal Grammars and Their Applications to Syntax Analysis

BNF Notation

- BNF stands for **Backus-Naur Form**. It is used to write a formal representation of a context-free grammar. It is also used to describe the syntax of a programming language.
- BNF notation is basically just a variant of a context-free grammar.

Formal Grammars and Their Applications to Syntax Analysis

Left side → definition

- Where leftside ∈ $(V_n \cup V_t)+$ and definition ∈ $(V_n \cup V_t)*$. In BNF, the leftside contains one non-terminal.
- We can define the several productions with the same leftside. All the productions are separated by a vertical bar symbol "|".

Yacc

- YACC stands for **Yet Another Compiler Compiler**.
- YACC provides a tool to produce a parser for a given grammar.
- YACC is a program designed to compile a LALR (1) grammar.
- It is used to produce the source code of the syntactic analyzer of the language produced by LALR (1) grammar.
- The input of YACC is the rule or grammar and the output is a C program.
- These are some points about YACC:

Yacc

- **Input: A CFG- file.y**
- **Output: A parser y.tab.c (yacc)**
- The output file "file.output" contains the parsing tables.
- The file "file.tab.h" contains declarations.
- The parser called the yyparse ().
- Parser expects to use a function called yylex () to get tokens.
- The basic operational sequence is as follows:

Context free grammar

- Context free grammar is a formal grammar which is used to generate all possible strings in a given formal language.
- Context free grammar G can be defined by four tuples as:

G= (V, T, P, S)

- **T** describes a finite set of terminal symbols.
- **V** describes a finite set of non-terminal symbols
- **P** describes a set of production rules
- **S** is the start symbol.

In CFG, the start symbol is used to derive the string. You can derive the string by repeatedly replacing a non-terminal by the right hand side of the production, until all non-terminal have been replaced by terminal symbols.

Context free grammar

L= {wcw^R | w € (a, b)*}

1.S → aSa

2.S → bSb

3.S → c

S ⇒ aSa

S ⇒ abSba

S ⇒ abbSbba

S ⇒ abbcbba

Context free grammar

Capabilities of CFG

- There are the various capabilities of CFG:
- Context free grammar is useful to describe most of the programming languages.
- If the grammar is properly designed then an efficient parser can be constructed automatically.
- Using the features of associatively & precedence information, suitable grammars for expressions can be constructed.
- Context free grammar is capable of describing nested structures like: balanced parentheses, matching begin-end, corresponding if-then-else's & so on.

Derivation

- Derivation is a sequence of production rules. It is used to get the input string through these production rules. During parsing we have to take two decisions. These are as follows:
- We have to decide the non-terminal which is to be replaced.
- We have to decide the production rule by which the non-terminal will be replaced.
- We have two options to decide which non-terminal to be replaced with production rule.
- Left-most Derivation
- In the left most derivation, the input is scanned and replaced with the production rule from left to right. So in left most derivatives we read the input string from left to right.

Derivation

1. S = S + S
2. S = S - S
3. S = a | b |c

Input : a - b + c

S = S + S

S = S - S + S

S = a - S + S

S = a - b + S

S = a - b + c

Derivation

Right-most Derivation

- In the right most derivation, the input is scanned and replaced with the production rule from right to left. So in right most derivatives we read the input string from right to left.

1. S = S + S
2. S = S - S
3. S = a | b |c

S = S - S

S = S - S + S

S = S - S + c

S = S - b + c

S = a - b + c

Capabilities of CFG

- - Defines recursive syntactic structures.
- - Parses arithmetic expressions, conditional statements, loops, etc.
- - Basis for modern parsers and compilers.