18CSC304J- Compiler Design

Subject handling Faculty
Mrs. G . ABIRAMI
AP (CSE)
SRMIST-KTR

WHY WE NEED COMPILER?

HOW IT IS WORKING?

Assembly & Machine Language

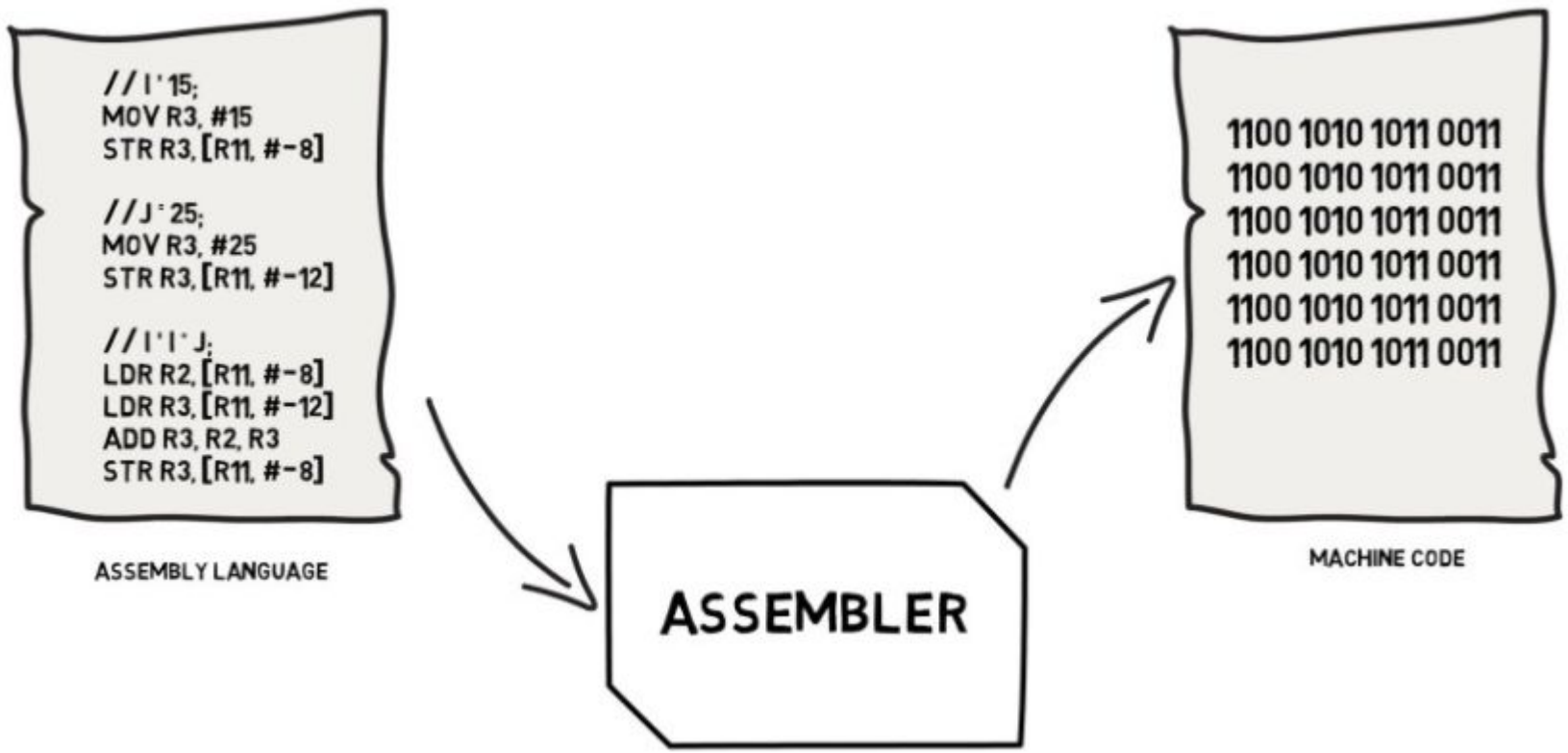
Download

Assembly Language

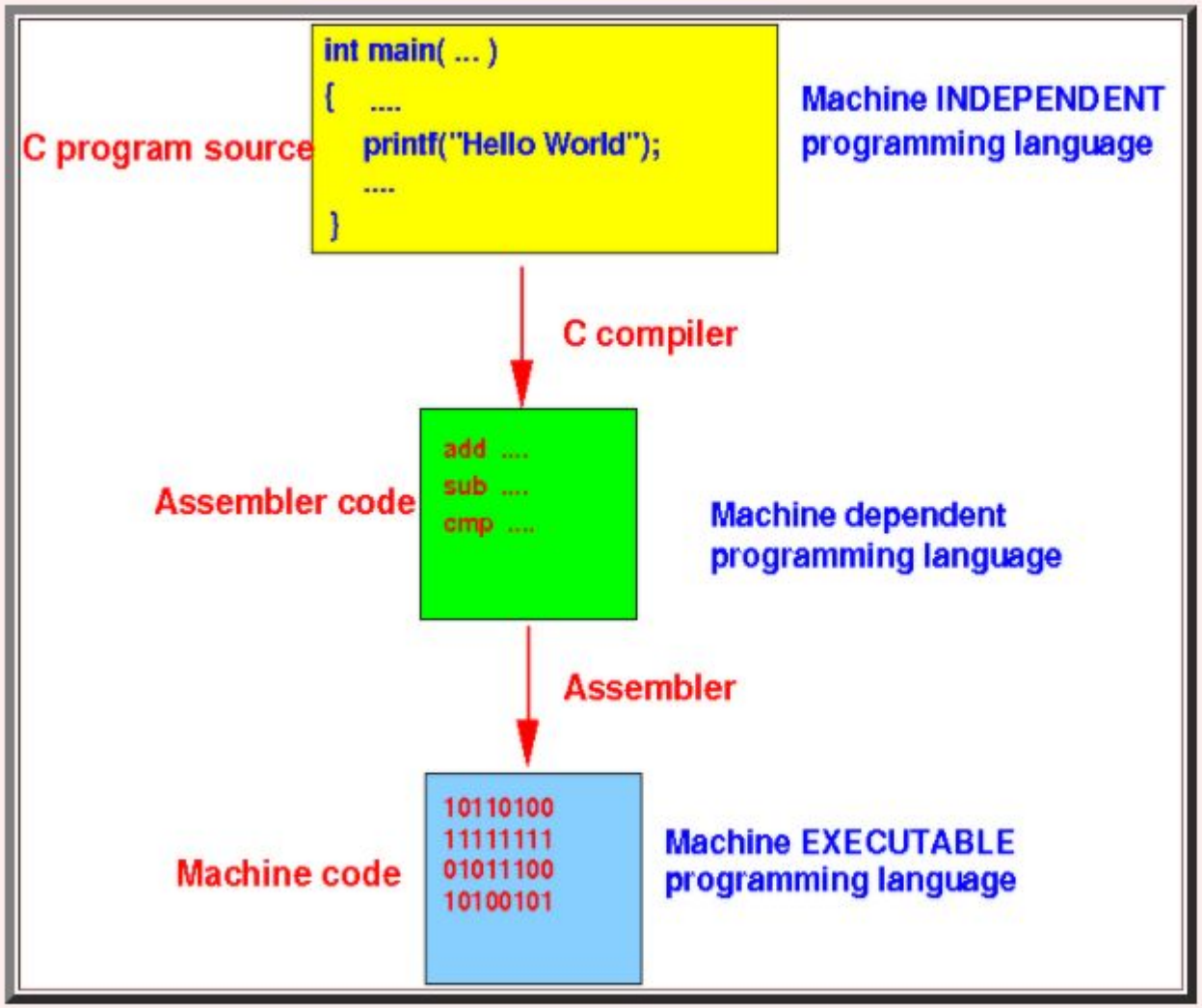
```
      ST 1,[801]
      ST 0,[802]
TOP:  BEQ [802],10,BOT
      INCR [802]
      MUL [801],2,[803]
      ST [803],[801]
      JMP   TOP
BOT:  LD A,[801]
      CALL PRINT
```

Machine Language

```
00100101 11010011
00100100 11010100
10001010 01001001 11110000
01000100 01010100
01001000 10100111 10100011
11100101 10101011 00000010
00101001
11010101
11010100 10101000
10010001 01000100
```


// I = 15;
MOV R3, #15
STR R3, [R11, #-8]
// J = 25;
MOV R3, #25
STR R3, [R11, #-12]
// I = I + J;
LDR R2, [R11, #-8]
LDR R3, [R11, #-12]
ADD R3, R2, R3
STR R3, [R11, #-8]
ASSEMBLY LANGUAGE
ASSEMBLER
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
1100 1010 1011 0011
MACHINE CODE

A C program (C is very similar to Java) is translated as follows:



First, the C compiler translates the C source program into an equivalent program written in an assembler (nemonic) programming language

Then an assembler translate the assembler code (program) into (binary) machine code (instruction)

Pre-requisite

- Basic knowledge of programming languages.
- Basic knowledge of FSA and CFG.
- Knowledge of a high programming language for the programming assignments.

Textbook:

Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman, "*Compilers: Principles, Techniques, and Tools*" Addison-Wesley, 1986.

Course Outline

- Introduction to Compiling
- Lexical Analysis
- Syntax Analysis
  - Context Free Grammars
  - Top-Down Parsing, LL Parsing
  - Bottom-Up Parsing, LR Parsing
- Syntax-Directed Translation
  - Attribute Definitions
  - Evaluation of Attribute Definitions
- Semantic Analysis, Type Checking
- Run-Time Organization
- Intermediate Code Generation

Course Code	18CSC304J	Course Name	COMPILER DESIGN	Course Category	C	Professional Core	L	T	P	C
							3	0	2	4

Pre-requisite Courses	18CSC301T	Co-requisite Courses	Nil	Progressive Courses	
Course Offering Department	Computer Science and Engineering	Data Book / Codes/Standards		Nil	

Course Learning Rationale (CLR):	*The purpose of learning this course is to:*	Learning			Program Learning Outcomes (PLO)														
		1	2	3	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		Level of Thinking (Bloom)	Expected Proficiency (%)	Expected Attainment (%)	Engineering Knowledge	Problem Analysis	Design & Development	Analysis, Design, Research	Modern Tool Usage	Society & Culture	Environment & Sustainability	Ethics	Individual & Team Work	Communication	Project Mgt. & Finance	Life Long Learning	PSO - 1	PSO - 2	PSO – 3
CLR-1 :	*Utilize the mathematics and engineering principles for the Design of Compilers*	3	80	70	H	H	H	H	M	L	L	L	M	M	L	H	H	H	H
CLR-2 :	*Acquire knowledge of Lexical Analyzer from a specification of a language's lexical rules*	3	85	75	H	H	H	H	M	L	L	L	M	M	L	H	H	H	H
CLR-3 :	*Acquire knowledge of Syntax Analyzer for parsing the sentences in a compiler grammar*	3	75	70	H	H	H	H	M	L	L	L	M	M	L	H	H	H	H
CLR-4 :	*Gain knowledge to translate a system into various intermediate codes*	3	85	80	H	H	H	H	M	L	L	L	M	M	L	H	H	H	H
CLR-5 :	*Analyze the methods of implementing a Code Generator for compilers*	3	85	75	H	H	H	H	M	L	L	L	M	M	L	H	H	H	H
CLR-6 :	*Analyze and Design the methods of developing a Code Optimizer*	3	80	70	H	H	H	H	M	L	L	L	M	M	L	H	H	H	H

Course Learning Outcomes (CLO):	*At the end of this course, learners will be able to:*
CLR-1 :	*Utilize the mathematics and engineering principles for the Design of Compilers*
CLR-2 :	*Acquire knowledge of Lexical Analyzer from a specification of a language's lexical rules*
CLR-3 :	*Acquire knowledge of Syntax Analyzer for parsing the sentences in a compiler grammar*
CLR-4 :	*Gain knowledge to translate a system into various intermediate codes*
CLR-5 :	*Analyze the methods of implementing a Code Generator for compilers*
CLR-6 :	*Analyze and Design the methods of developing a Code Optimizer*

Duration (hour)		15	15	15	15	15
S-1	SLO-1	*Compilers – Analysis of the source program*	*Syntax Analysis Definition - Role of parser*	*Bottom Up Parsing*	*Intermediate Code Generation*	*Code optimization*
	SLO-2	*Phases of a compiler – Cousins of the Compiler*	*Lexical versus Syntactic Analysis*	*Reductions*	*Intermediate Languages - prefix - postfix*	*Introduction– Principal Sources of Optimization*
S-2	SLO-1	*Grouping of Phases – Compiler construction tools*	*Representative Grammars*	*Handle Pruning*	*Quadruple - triple - indirect triples Representation*	*Function Preserving Transformation*
	SLO-2	*Lexical Analysis – Role of Lexical Analyzer*	*Syntax Error Handling*	*Shift Reduce Parsing*	*Syntax tree- Evaluation of expression - three-address code*	*Loop Optimization*
S-3	SLO-1	*Input Buffering*	*Elimination of Ambiguity, Left Recursion*	*Problems related to Shift Reduce Parsing*	*Synthesized attributes – Inherited attributes*	*Optimization of basic Blocks*
	SLO-2	*Specification of Tokens*	*Left Factoring*	*Conflicts During Shift Reduce Parsing*	*Intermediate languages – Declarations*	*Building Expression of DAG*
S 4-5	SLO-1	*Lab 1 - Implementation of Lexical Analyzer*	*Lab 4Elimation of Ambiguity, Left Recursion and Left Factoring*	*Lab 7 - Shift Reduce Parsing*	*Lab 10-Intermediate code generation – Postfix, Prefix*	*Lab 13 Implementation of DAG*
	SLO-2					
S-6	SLO-1	*Finite automation - deterministic*	*Top down parsing*	*LR Parsers- Why LR Parsers*	*Assignment Statements*	*Peephole Optimization*
	SLO-2	*Finite automation - non deterministic*	*Recursive Descent Parsing, back tracking*	*Items and LR(0) Automaton, Closure of Item Sets,*	*Boolean Expressions, Case Statements*	*Basic Blocks, Flow Graphs*
S-7	SLO-1	*Transition Tables*	*Computation of FIRST*	*LR Parsing Algorithm*	*Back patching – Procedure calls*	*Next -Use Information*

UNIT -1

Duration (hour)		15
S-1	SLO-1	*Compilers – Analysis of the source program*
	SLO-2	*Phases of a compiler – Cousins of the Compiler*
S-2	SLO-1	*Grouping of Phases – Compiler construction tools*
	SLO-2	*Lexical Analysis – Role of Lexical Analyzer*
S-3	SLO-1	*Input Buffering*
	SLO-2	*Specification of Tokens*
S 4-5	SLO-1	*Lab 1 - Implementation of Lexical Analyzer*
	SLO-2	
S-6	SLO-1	*Finite automation - deterministic*
	SLO-2	*Finite automation - non deterministic*
S-7	SLO-1	*Transition Tables*

TRANSLATOR

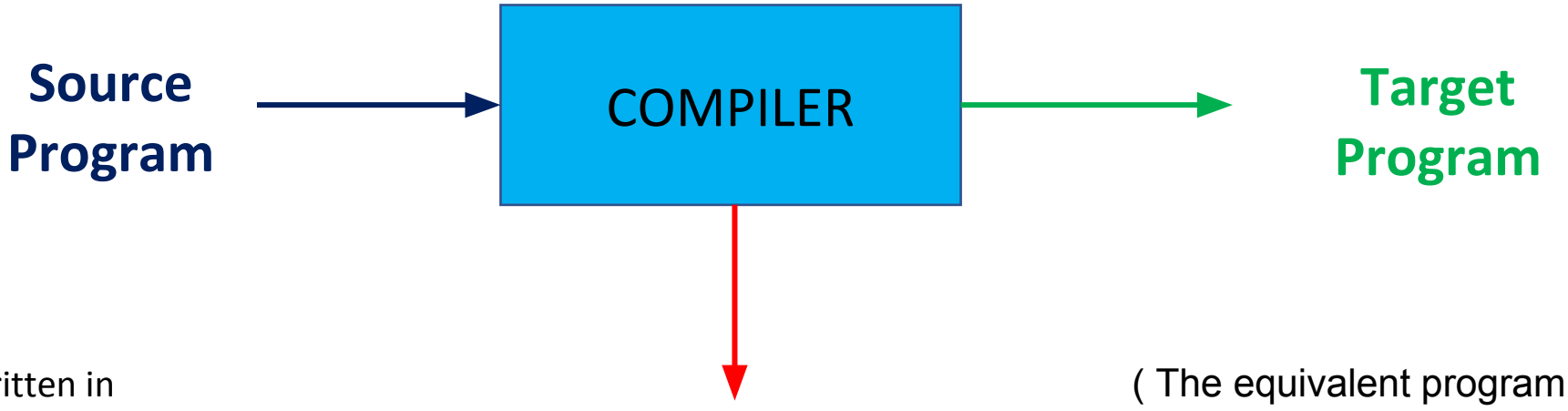
Translate high level language into equivalent machine language

Types

- Compiler
- Interpreter
- Assembler

COMPILERS

- A **compiler** is a program takes a program written in a source language and translates it into an equivalent program in a target language.



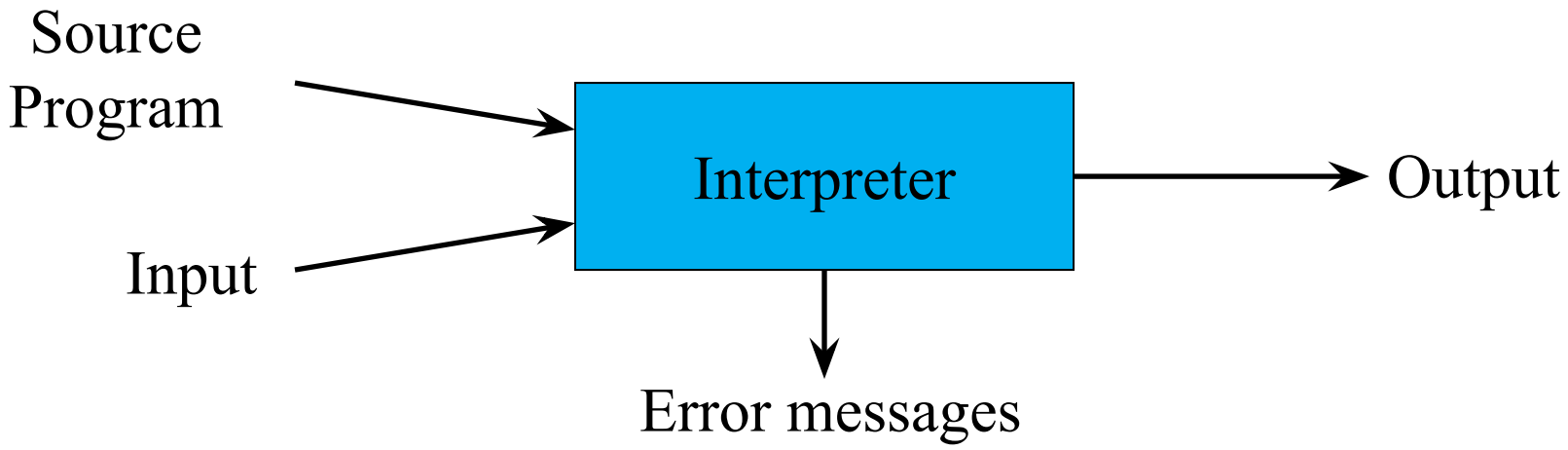
(A program written in a high-level programming language)

(The equivalent program in machine code – relocatable object file)

(Take correct actions to remove the errors for further compilation process)

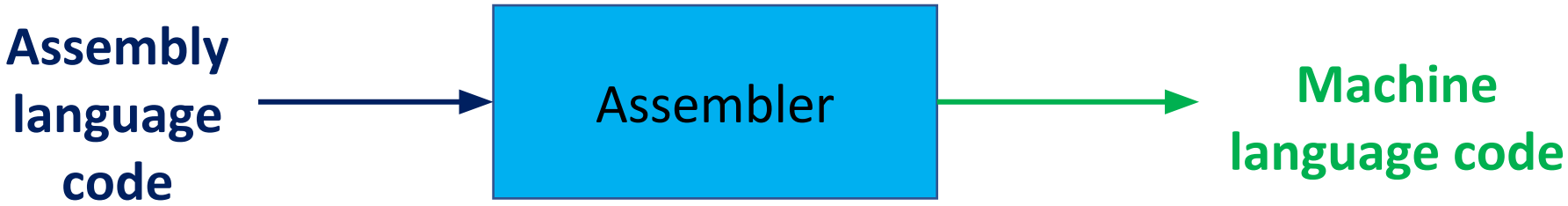
INTERPRETER

- Translates line by line
- Better error diagnostics



ASSEMBLER

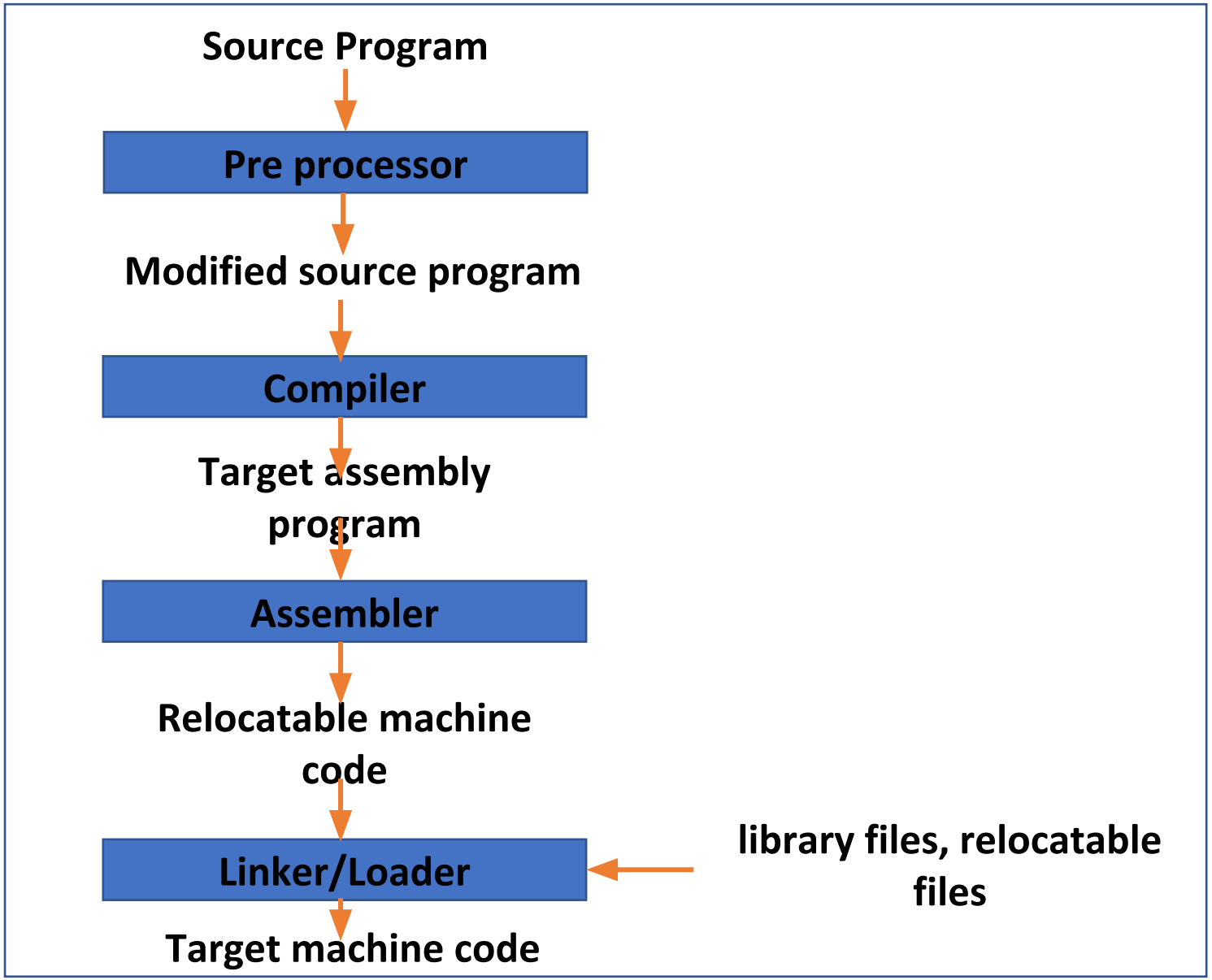
Assembler is a translator which is used to translate the assembly language code into machine language code

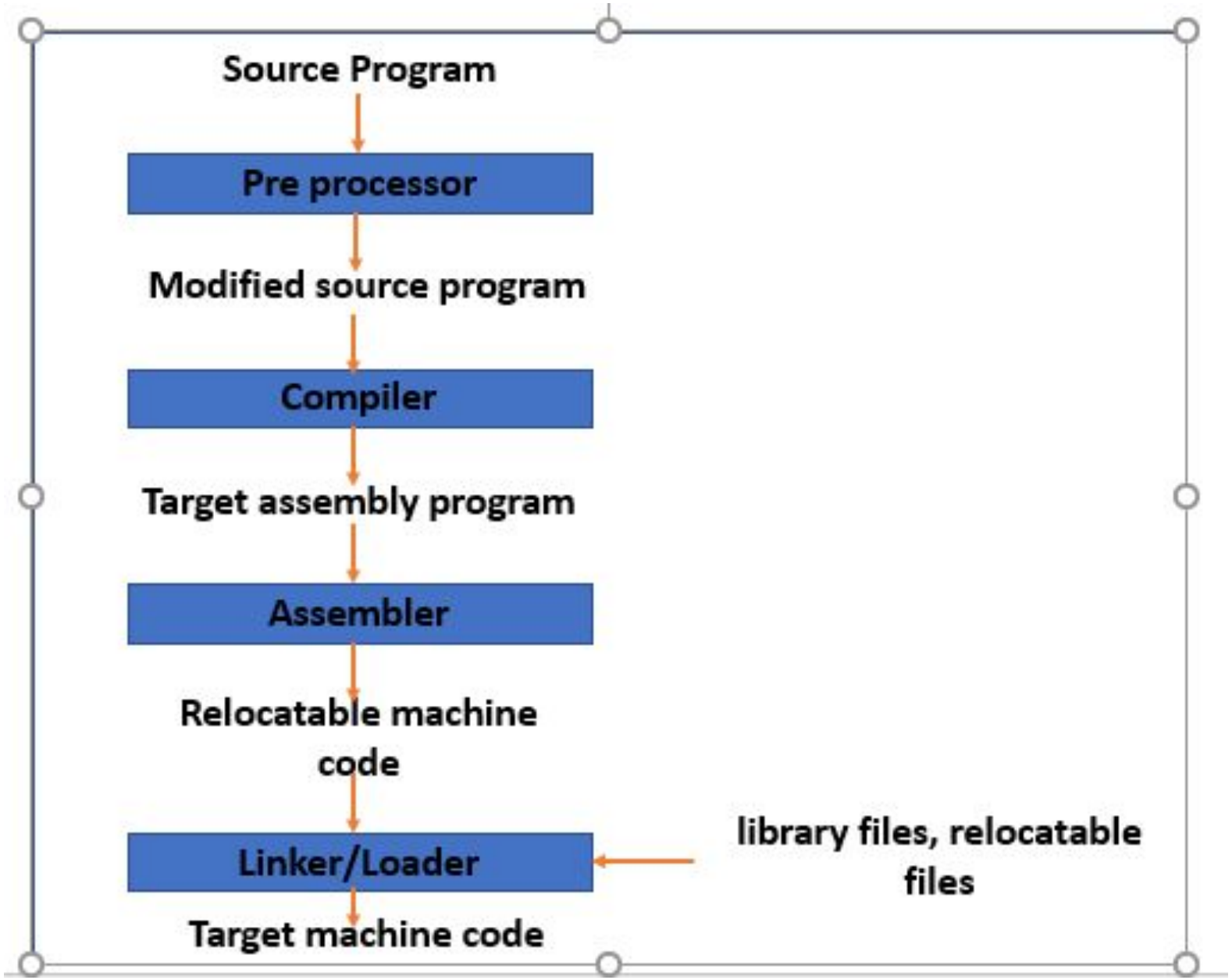


Language Processor

- **Language Processors** An integrated software development environment includes many different kinds of language processors such as compilers, interpreters, assemblers, linkers, loaders, debuggers, profilers.

Language processor




Source Program
Pre processor
Modified source program
Compiler
Target assembly program
Assembler
Relocatable machine code
Linker/Loader
library files, relocatable files
Target machine code

Pre-processor

A source program may be divided into modules stored in separate files. The task of collecting the source program is entrusted to a separate program called pre-processor. It may also expand macros into source language statement.

Compiler

Compiler is a program that takes source program as input and produces assembly language program as output.

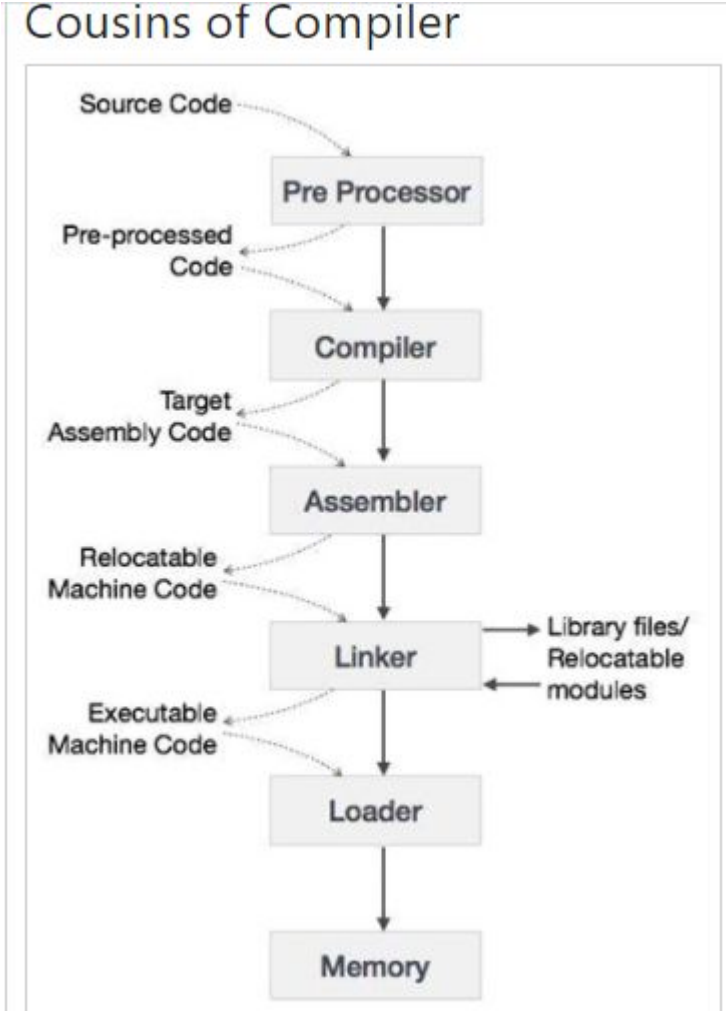
Assembler

Assembler is a program that converts assembly language program into machine language program. It produces re-locatable machine code as its output.

Loader and link-editor

- The re-locatable machine code has to be linked together with other re-locatable object files and library files into the code that actually runs on the machine.
- The linker resolves external memory addresses, where the code in one file may refer to a location in another file.
- The loader puts together the entire executable object files into memory for execution.

Cousins of the Compiler



1) Preprocessor

It converts the HLL (high level language) into pure high level language. It includes all the header files and also evaluates if any macro is included. It is the optional because if any language which does not support `#include` and macro preprocessor is not required.

2) Compiler

It takes pure high level language as a input and convert into assembly code.

3) Assembler

It takes assembly code as an input and converts it into assembly code.

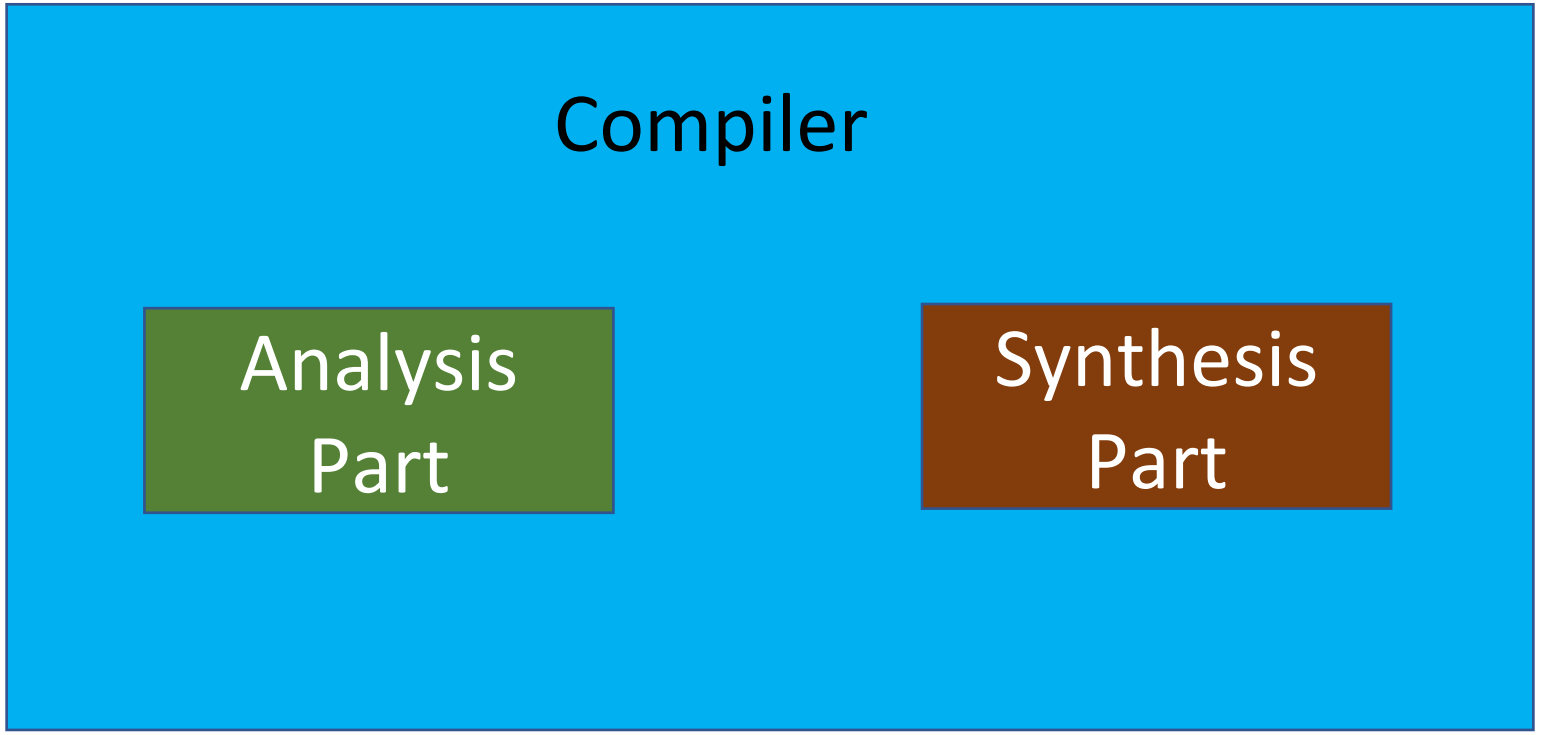
4) Linking and loading

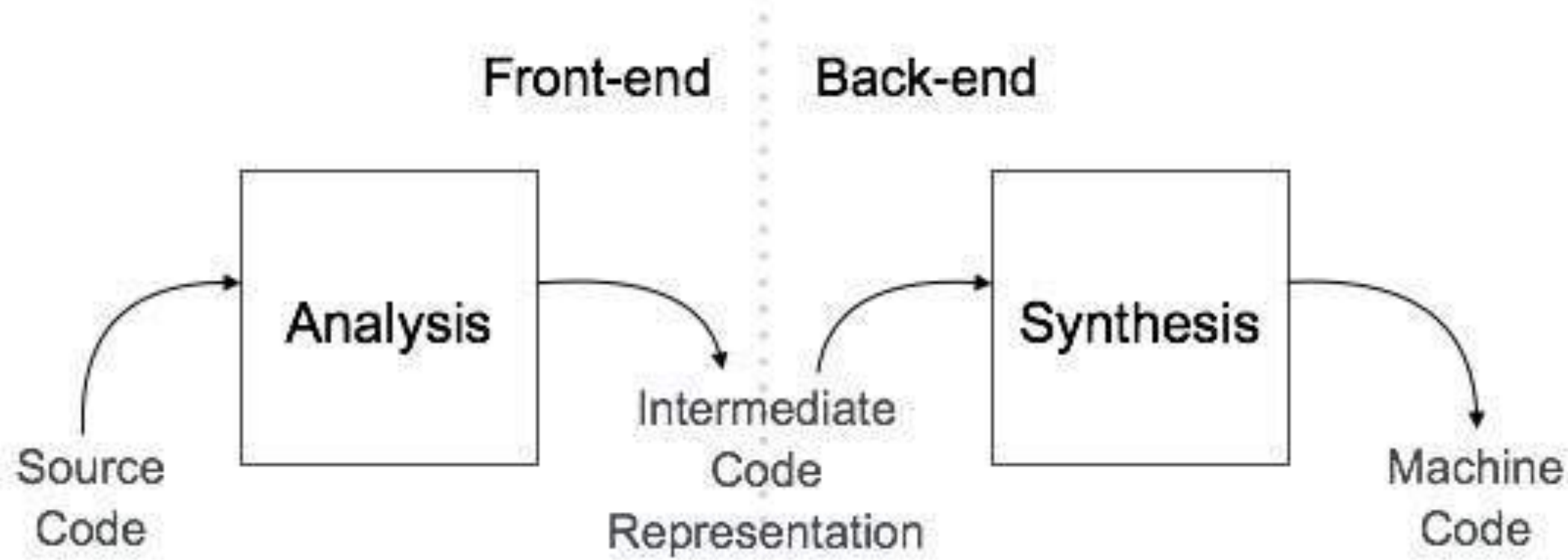
It has four functions

1. **Allocation:**
   It means get the memory portions from operating system and storing the object data.
2. **Relocation:**
   It maps the relative address to the physical address and relocating the object code.
3. **Linker:**
   It combines all the executable object module to pre single executable file.
4. **Loader:**
   It loads the executable file into permanent storage.

STRUCTURE OF THE COMPILER

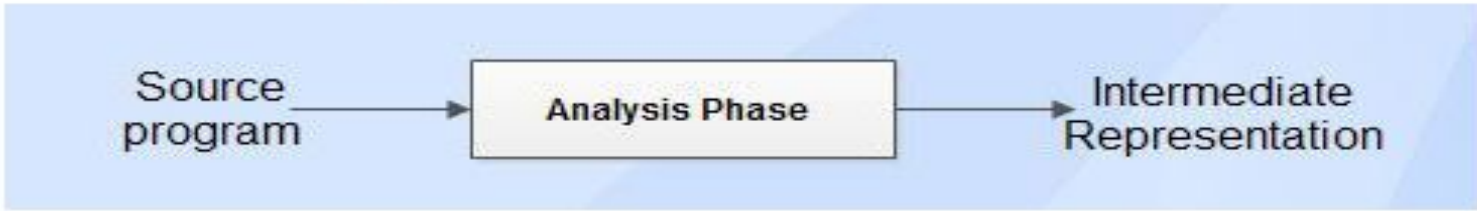
Two Parts:




Front-end
Back-end
Analysis
Synthesis
Source Code
Intermediate Code Representation
Machine Code

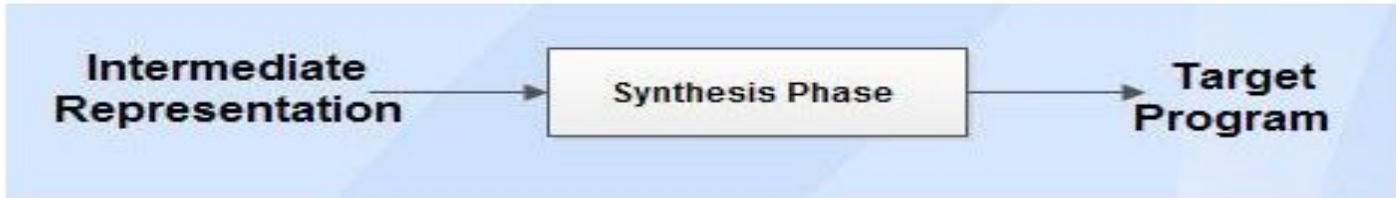
Analysis part

- Analysis part breaks the source program into constituent pieces and imposes a grammatical structure on them which further uses this structure to create an intermediate representation of the source program.
- It is also termed as front end of compiler.
- Information about the source program is collected and stored in a data structure called symbol table.



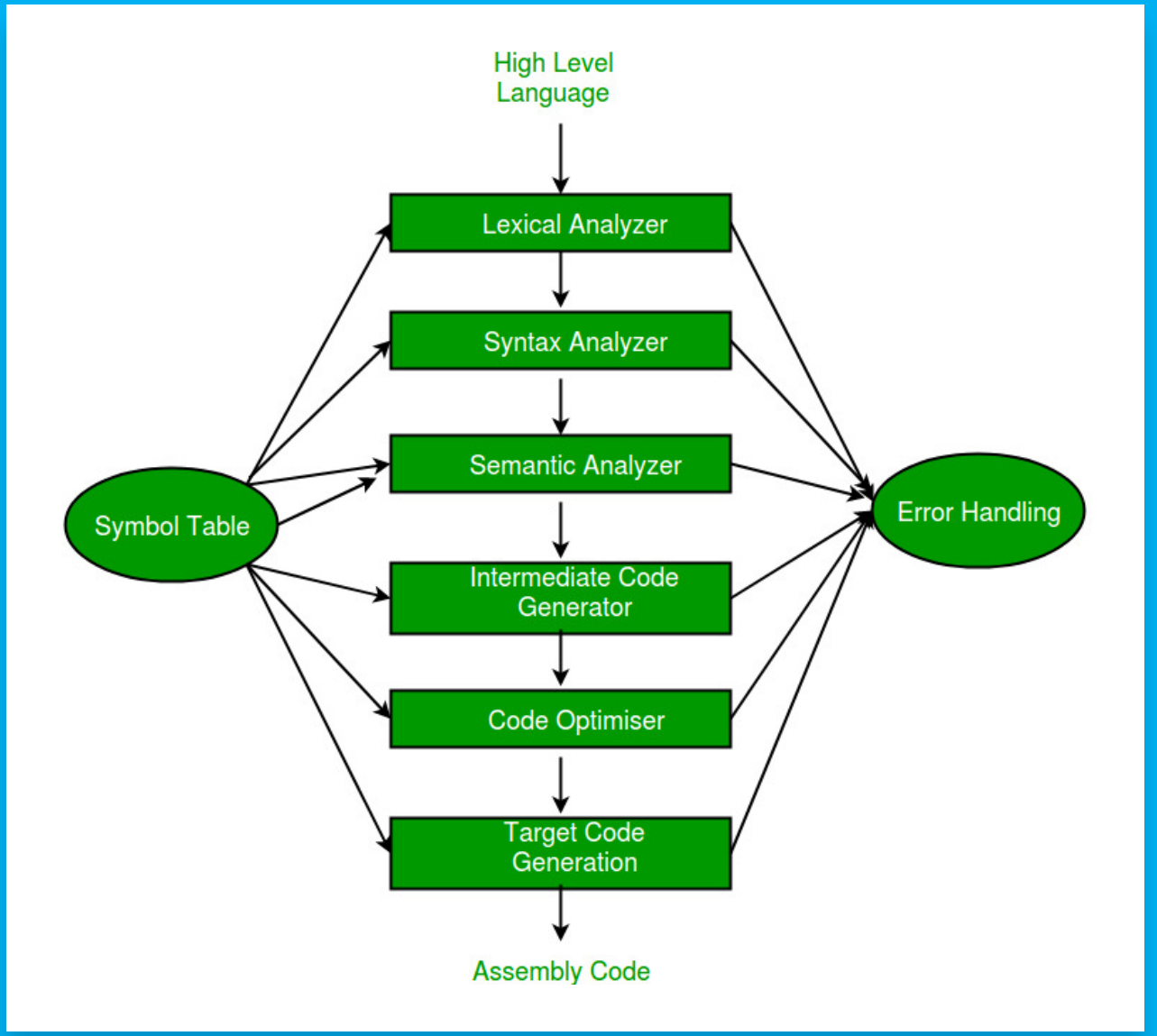
Synthesis part

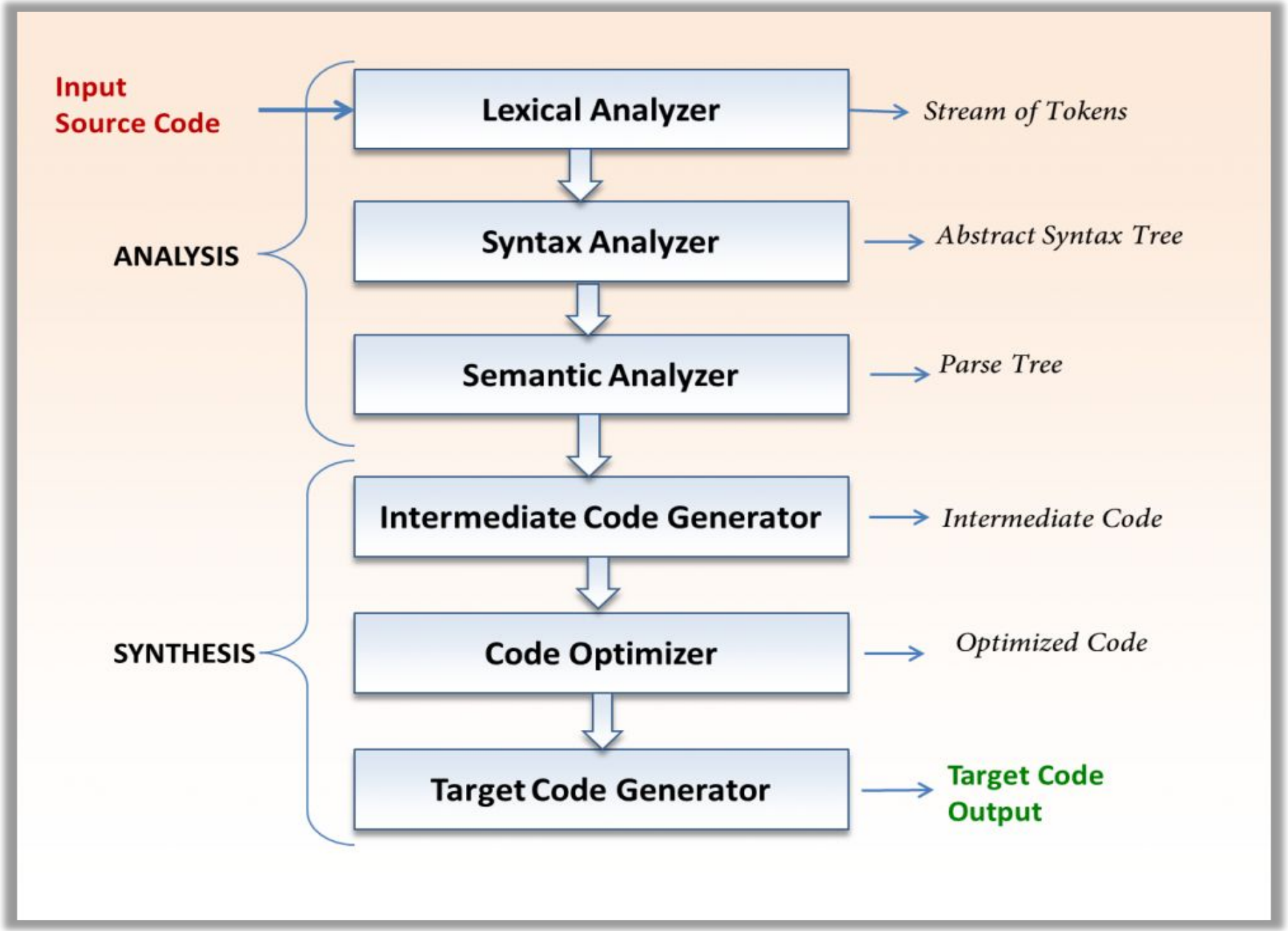
- Synthesis part takes the intermediate representation as input and transforms it to the target program.
- It is also termed as back end of compiler.



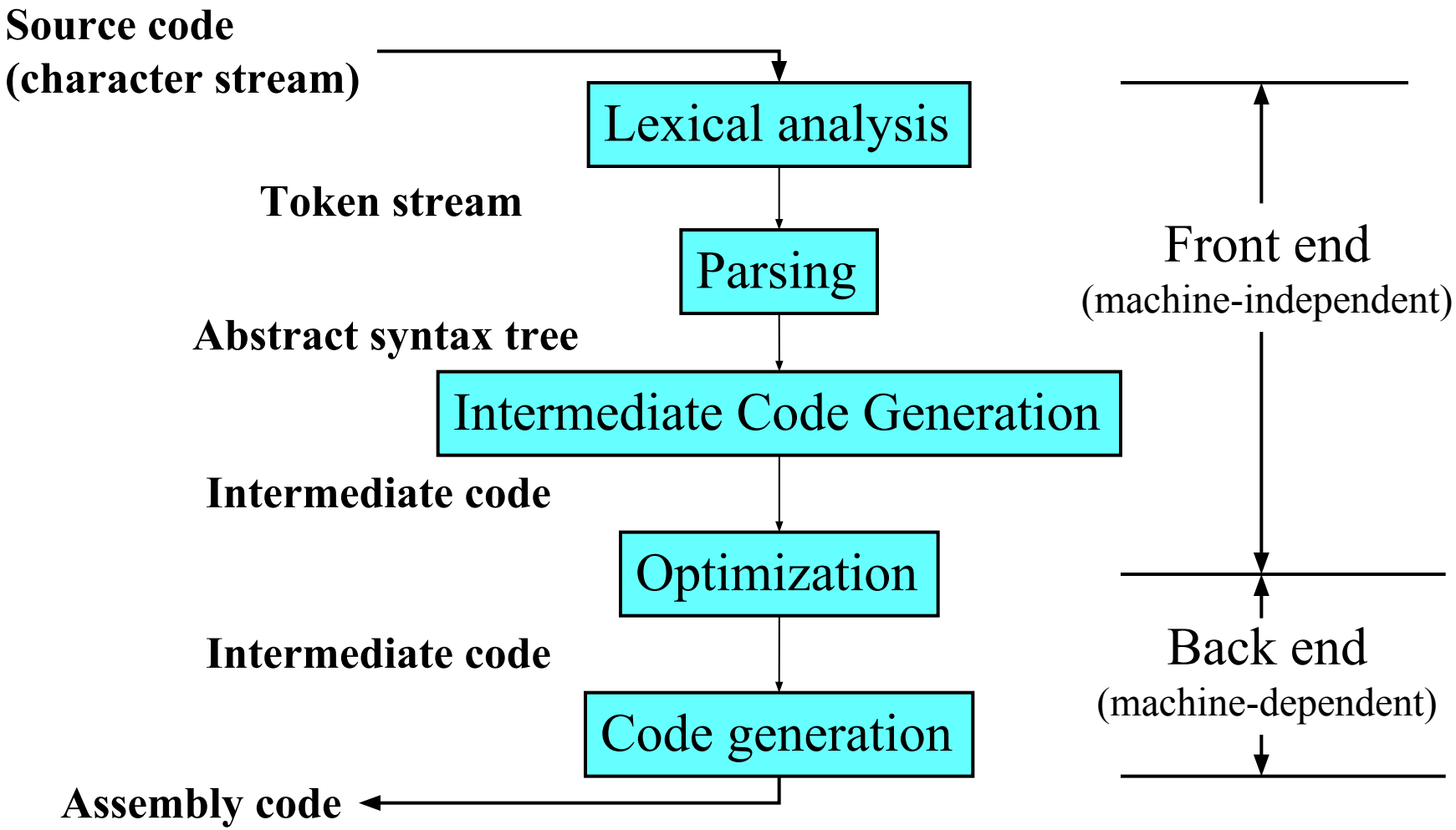
The design of compiler can be decomposed into several phases, each of which converts one form of source program into another.

Phases of the Compiler



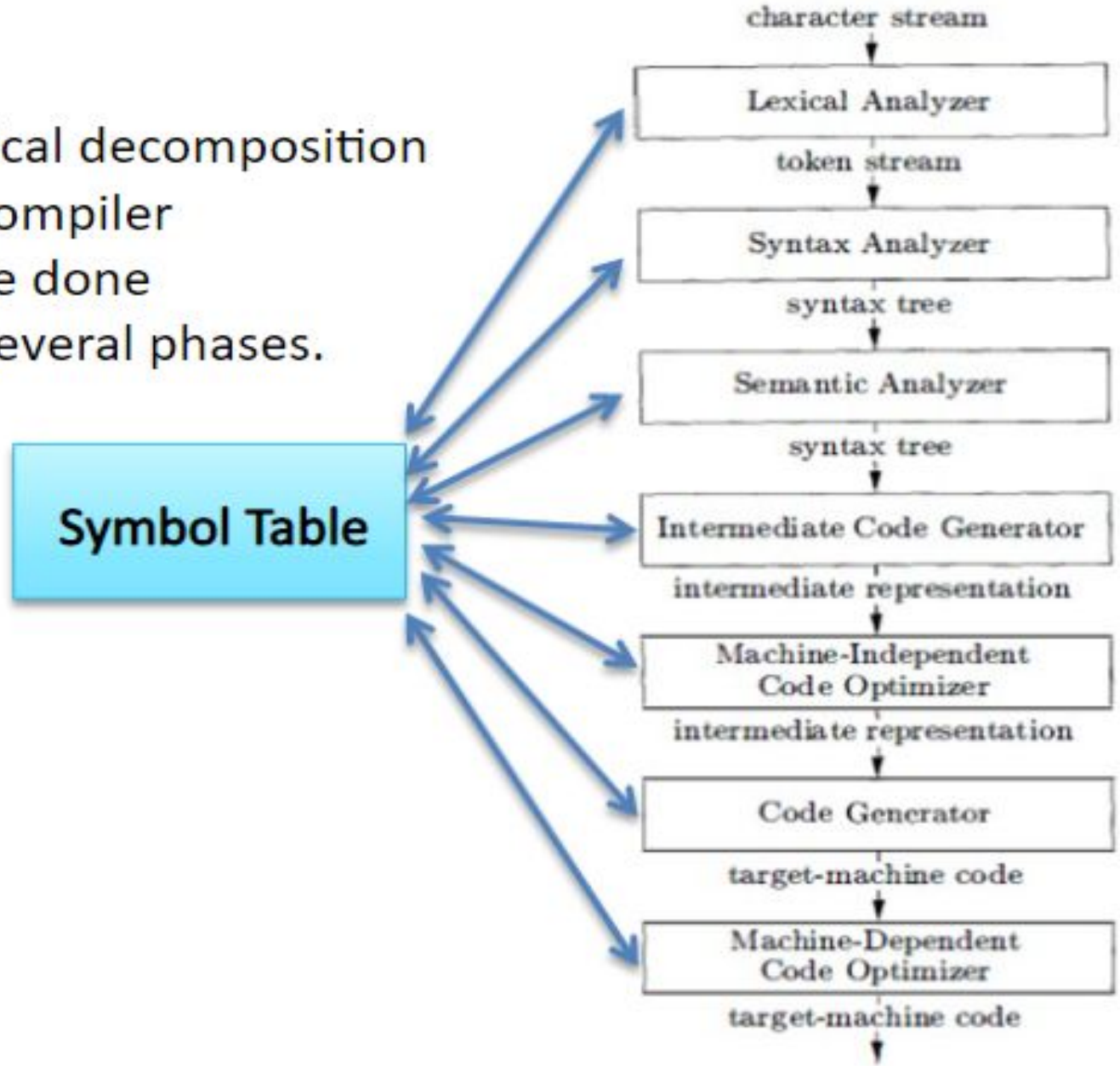

Input
Source Code
ANALYSIS
Lexical Analyzer
Stream of Tokens
Syntax Analyzer
Abstract Syntax Tree
Semantic Analyzer
Parse Tree
SYNTHESIS
Intermediate Code Generator
Intermediate Code
Code Optimizer
Optimized Code
Target Code Generator
Target Code
Output

Standard Compiler Structure

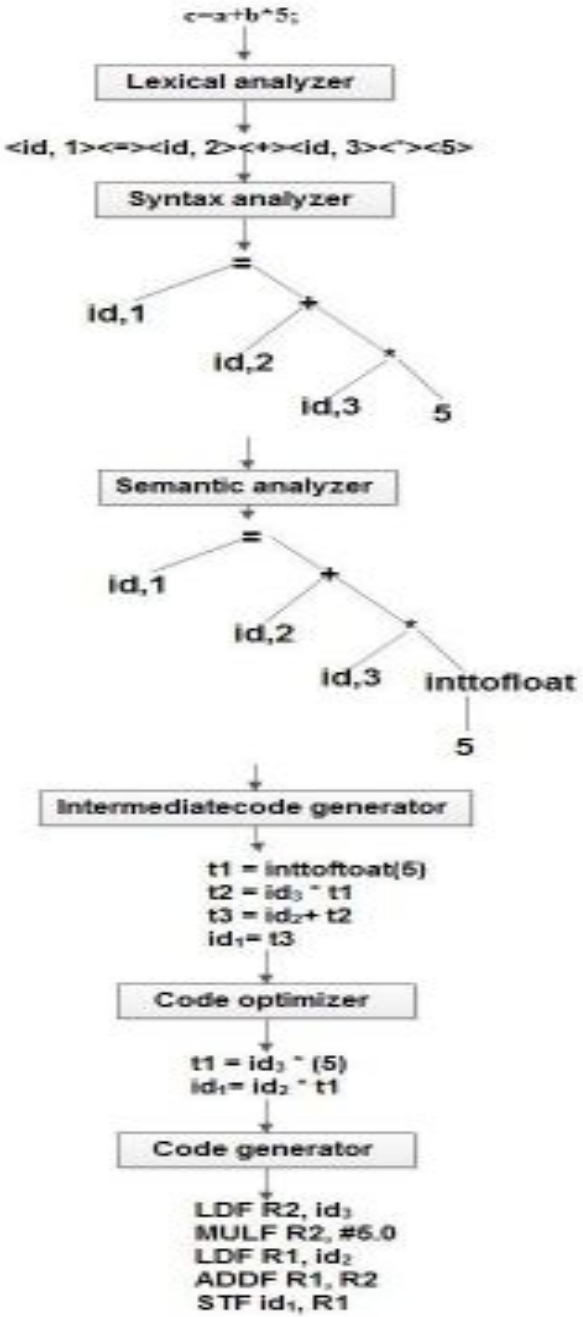


Compilation Phases

- A typical decomposition of a compiler can be done into several phases.



Translation of an Assignment Statement



LEXICAL ANALYSIS

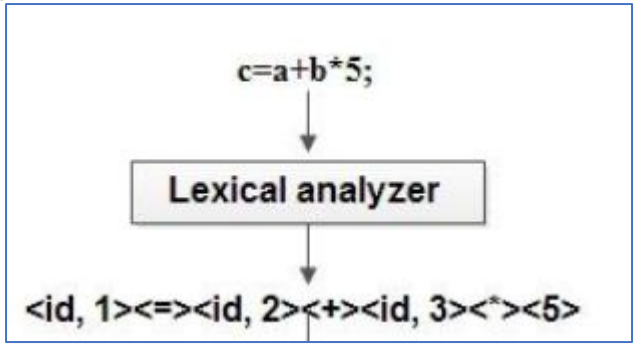
Lexical analysis

- Lexical analysis is the first phase of compiler which is also termed as **scanning.**
- Source program is scanned to read the stream of characters and those characters are grouped to form a sequence called **lexemes** which produces token as output.
- **Token:** Token is a sequence of characters that represent lexical unit, which matches with the pattern, such as **keywords, operators, identifiers etc.**
- **Lexeme:** Lexeme is instance of a token i.e., group of characters forming a token. ,
- **Pattern:** Pattern **describes the rule** that the lexemes of a token takes. It is the structure that must be matched by strings.
- Once a token is generated the corresponding entry is made in the symbol table.

Input: stream of characters

Output: Token

Token Template: ***<token-name, attribute-value>***



Example : c=a+b*5;

Lexemes and tokens

Lexemes	Tokens
c	identifier
=	assignment symbol
a	identifier
+	+ (addition symbol)
b	identifier
*	* (multiplication symbol)
5	5 (number)

Hence, <id, 1><=>< id, 2>< +><id, 3 >< * >< 5>

The primary functions of Lexical Phase:

- Identify the lexical units in a source code
- Classify lexical units into classes like constants, reserved words, and enter them in different tables. It will Ignore comments in the source program
- Identify token which is not a part of the language

Example:

x = y + 10

Tokens

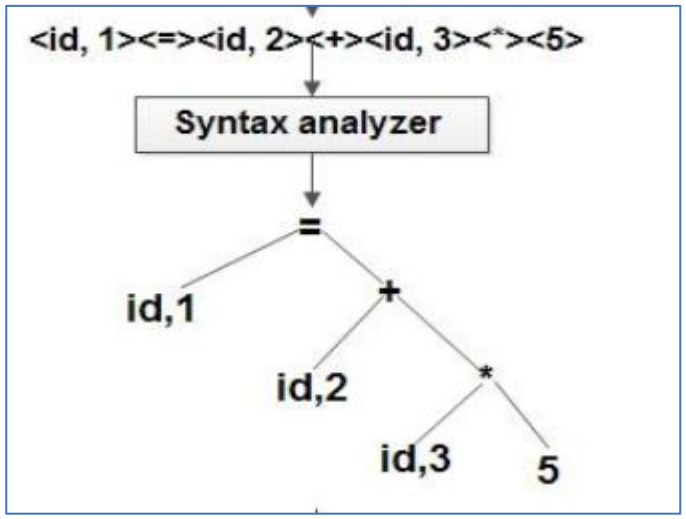
X	identifier
=	Assignment operator
Y	identifier
+	Addition operator
10	Number

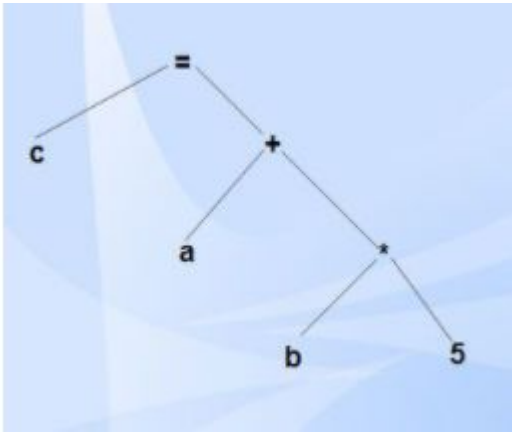
SYNTAX ANALYSIS

Syntax Analysis

- Syntax analysis is the second phase of compiler which is also called as **parsing.**
- Constructing the parse tree with the help of tokens. It also determines the structure of source language and grammar or syntax of the language.
- Parser converts the tokens produced by lexical analyser into a tree like representation called parse tree.
- Syntax tree is a compressed representation of the parse tree in which the operators appear as interior nodes and the operands of the operator are the children of the node for that operator.

Input: *Tokens*
Output: *Syntax tree*





Here, is a list of tasks performed in this phase:

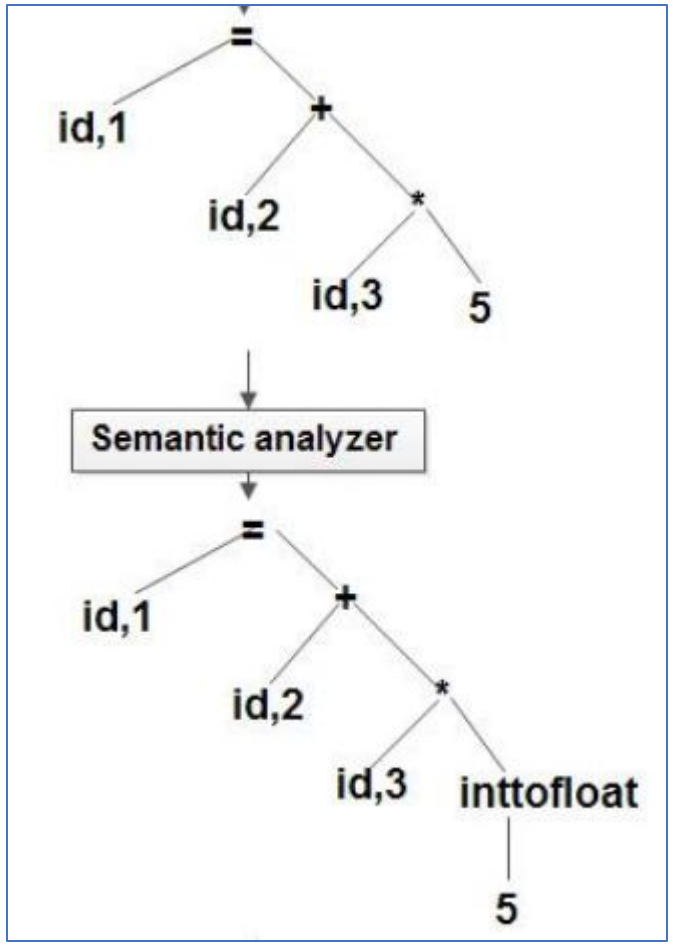
- Obtain tokens from the lexical analyser
- Checks if the expression is syntactically correct or not
- Report all syntax errors
- Construct a hierarchical structure which is known as a parse tree

SEMANTIC ANALYSIS

Semantic Analysis

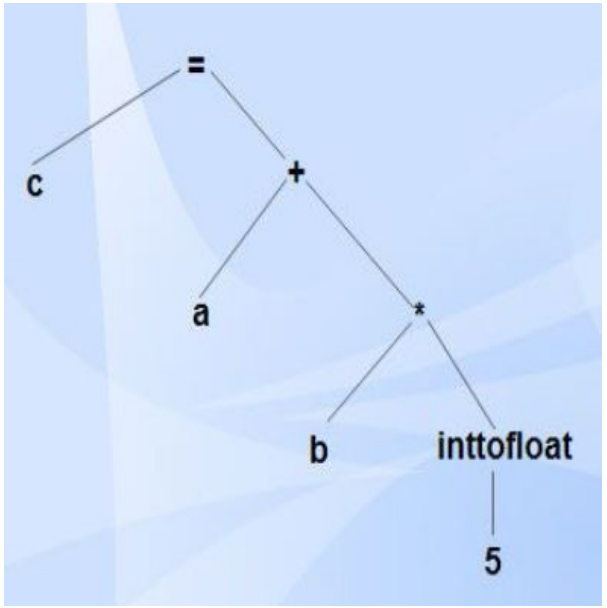
Semantic Analysis

- Semantic analysis is the third phase of compiler.
- It checks for the semantic consistency.
- Type information is gathered and stored in symbol table or in syntax tree.
- Performs type checking.



❖ **Type conversion and Type Coercion**

EXAMPLE



INTERMEDIATE CODE GENERATOR

Intermediate Code Generator

Intermediate Code Generation

• Intermediate code generation produces intermediate representations for the source program which are of the following forms:

- o Postfix notation
- o Three address code
- o Syntax tree

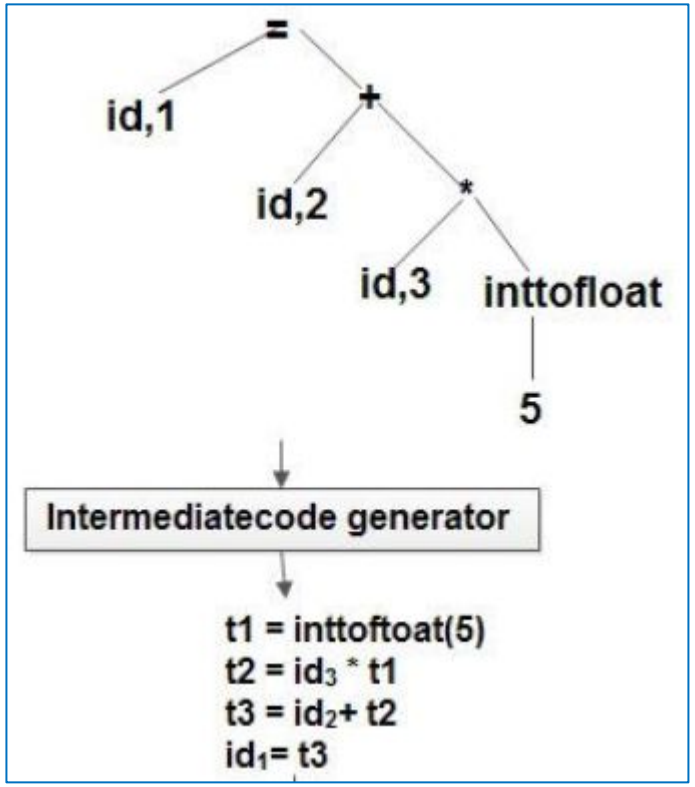
Most commonly used form is the three address code.

t_1 = inttofloat (5)

t_2 = id_3* tl

t_3 = id_2 + t_2

id_1 = t_3



Properties of intermediate code

• It should be easy to produce.

• It should be easy to translate into target program.

CODE OPTIMIZATION

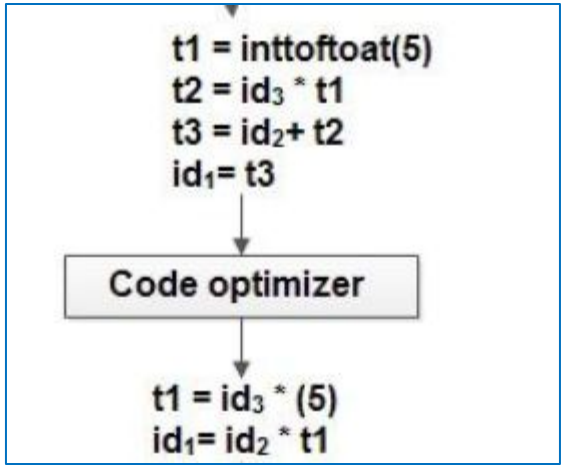
Code Optimization

Code Optimization

- Code optimization phase gets the intermediate code as input and produces optimized intermediate code as output.
- It results in faster running machine code.
- It can be done by reducing the number of lines of code for a program.
- This phase reduces the redundant code and attempts to improve the intermediate code so that faster-running machine code will result.
- During the code optimization, the result of the program is not affected.
- To improve the code generation, the optimization involves
  - o Deduction and removal of dead code (unreachable code).
  - o Calculation of constants in expressions and terms.
  - o Collapsing of repeated expression into temporary string.
  - o Loop unrolling.
  - o Moving code outside the loop.
  - o Removal of unwanted temporary variables.

$t_1 = id_3 * 5.0$

$id_1 = id_2 + t_1$



CODE GENERATOR

Code Generator

Code Generation

- Code generation is the final phase of a compiler.
- It gets input from code optimization phase and produces the target code or object code as result.
- Intermediate instructions are translated into a sequence of machine instructions that perform the same task.
- The code generation involves
  - o Allocation of register and memory.
  - o Generation of correct references.
  - o Generation of correct data types.
  - o Generation of missing code.

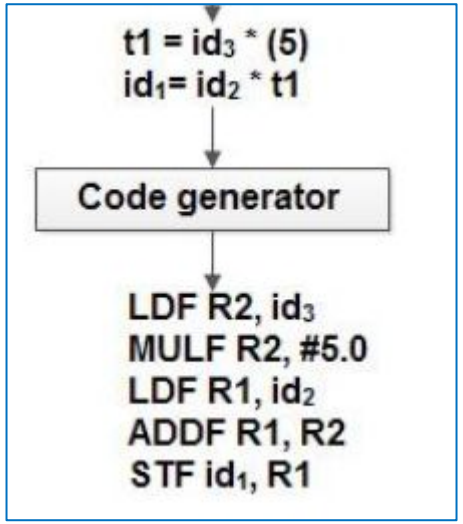
LDF R_2, id_3

MULF R_2, # 5.0

LDF R_1, id_2

ADDF R_1, R_2

STF id_1, R_1



Symbol Table

Symbol Table Management

- Symbol table is used to store all the information about identifiers used in the program.
- It is a data structure containing a record for each identifier, with fields for the attributes of the identifier.
- It allows finding the record for each identifier quickly and to store or retrieve data from that record.
- Whenever an identifier is detected in any of the phases, it is stored in the symbol table.

Example

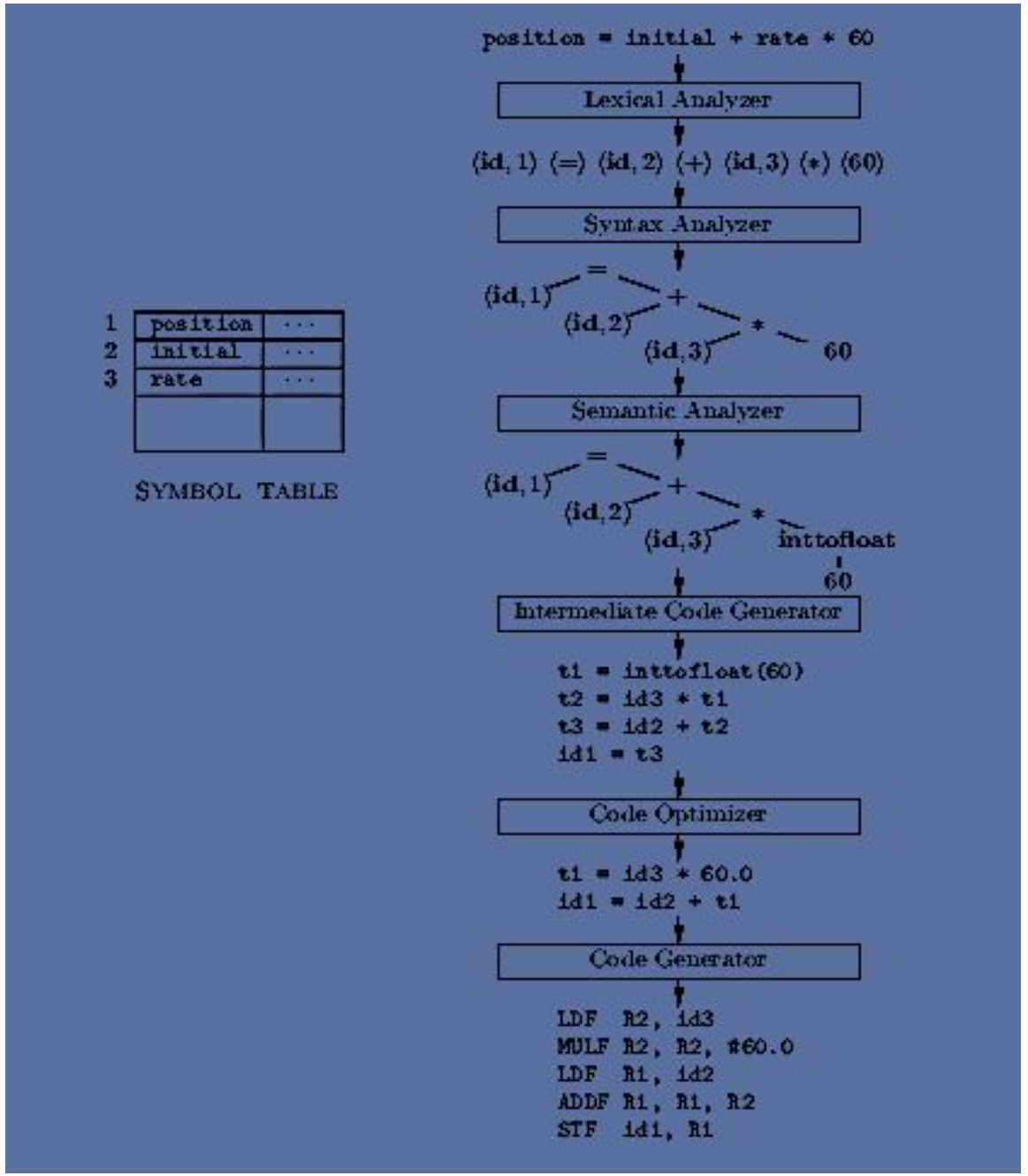
int a, b; float c; char z;

Symbol name	Type	Address
a	Int	1000
b	Int	1002
c	Float	1004
z	char	1008

Example

Translation of Statement

EXERCISE NO 1



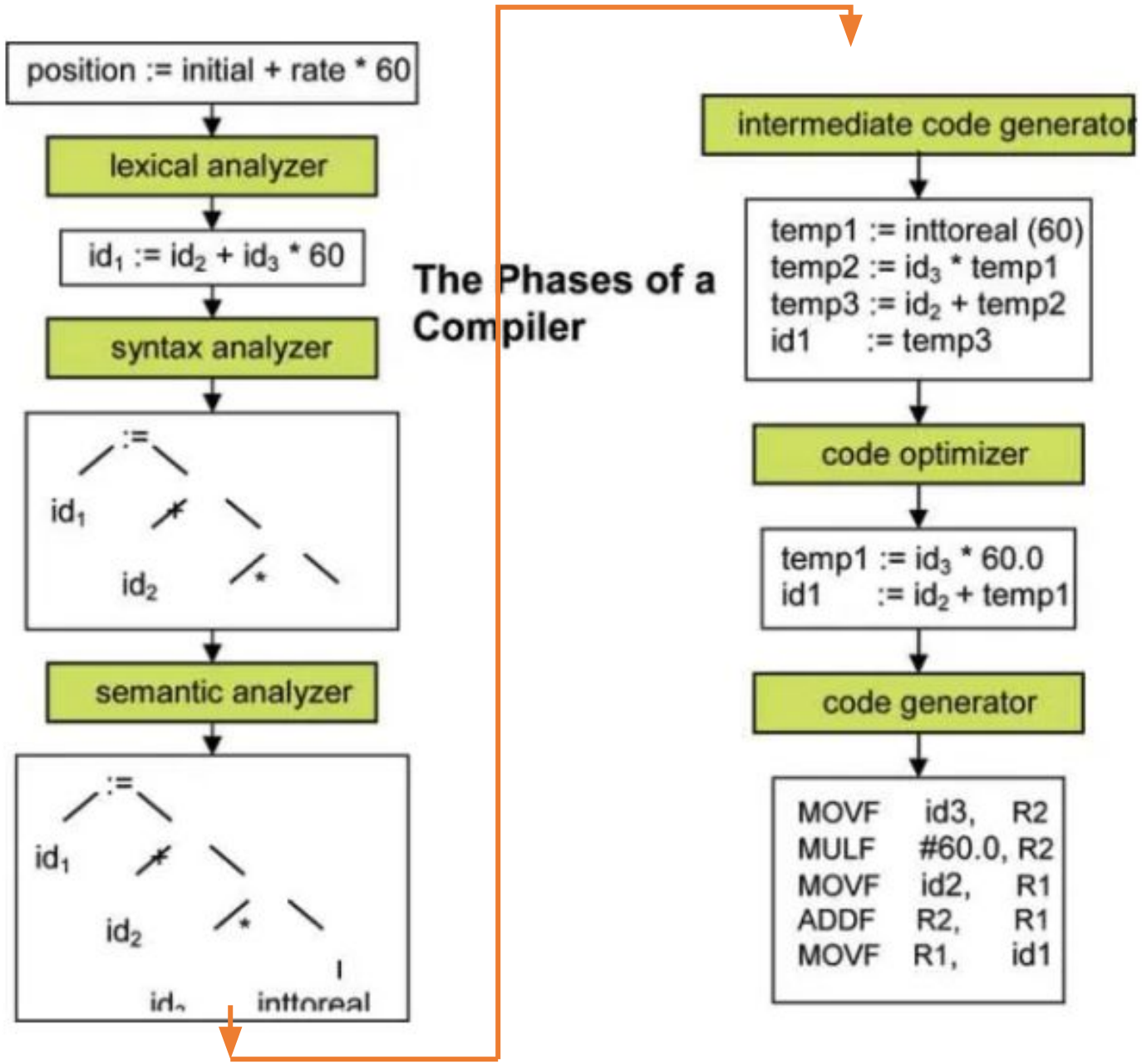
Q1

Phases of a compiler

Translation of an assignment statement

EXERCISE NO 1

Position = initial +rate *60


position := initial + rate * 60
lexical analyzer
id1 := id2 + id3 * 60
syntax analyzer
:=
id1
*
id2
*
semantic analyzer
:=
id1
*
id2
*
inttoreal
The Phases of a Compiler
intermediate code generator
temp1 := inttoreal (60)
temp2 := id3 * temp1
temp3 := id2 + temp2
id1 := temp3
code optimizer
temp1 := id3 * 60.0
id1 := id2 + temp1
code generator
MOVF id3, R2
MULF #60.0, R2
MOVF id2, R1
ADDF R2, R1
MOVF R1, id1

Error Handling

Error Handling

Each phase can encounter errors. After detecting an error, a phase must handle the error so that compilation can proceed.

• In lexical analysis, errors occur in separation of tokens.

• In syntax analysis, errors occur during construction of syntax tree.

• In semantic analysis, errors may occur at the following cases:

(i) When the compiler detects constructs that have right syntactic structure but no meaning

(ii) During type conversion.

• In code optimization, errors occur when the result is affected by the optimization. In code generation, it shows error when code is missing etc.

Figure illustrates the translation of source code through each phase, considering the statement

c =*a*+ *b* * 5.

Error Encountered in Different Phases

Each phase can encounter errors. After detecting an error, a phase must some how deal with the error, so that compilation can proceed. A program may have the following kinds of errors at various stages:

❑ Lexical Errors

It includes incorrect or misspelled name of some identifier i.e., identifiers typed incorrectly.

❑ Syntactical Errors

- It includes missing semicolon or unbalanced parenthesis.
- When an error is detected, it must be handled by parser to enable the parsing of the rest of the input.
- In general, errors may be expected at various stages of compilation but most of the errors are syntactic errors and hence the parser should be able to detect and report those errors in the program.

The goals of error handler in parser are:

- Report the presence of errors clearly and accurately.
- Recover from each error quickly enough to detect subsequent errors.
- Add minimal overhead to the processing of correcting programs.

There are four common error-recovery strategies that can be implemented in the parser to deal with errors in the code.

o Panic mode.

o Statement level.

o Error productions.

o Global correction.

❑ Semantical Errors

These errors are a result of incompatible value assignment. The semantic errors that the semantic analyzer is expected to recognize are:

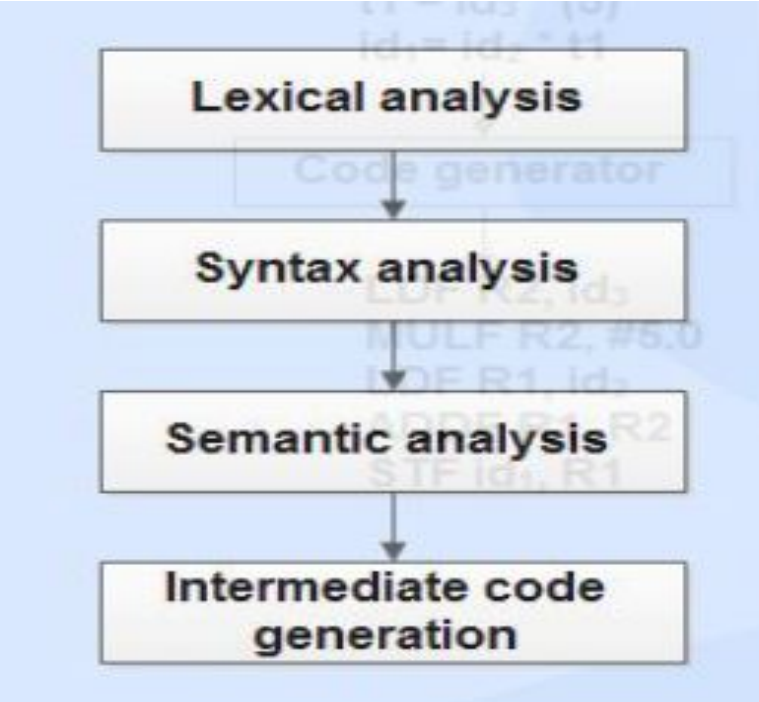
- Type mismatch.
- Undeclared variable.
- Reserved identifier misuse.
- Multiple declaration of variable in a scope.
- Accessing an out of scope variable.
- Actual and formal parameter mismatch.

Logical errors

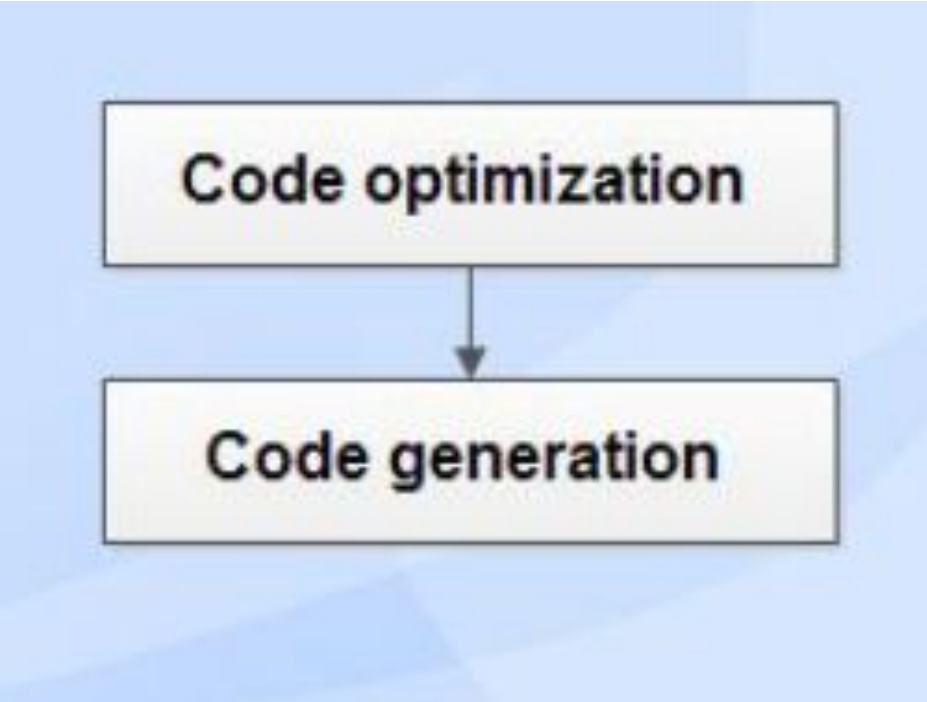
These errors occur due to not reachable code-infinite loop

Grouping of Phases

Front End



Back End



Front End

• Front end comprises of phases which are dependent on the input (source language) and independent on the target machine (Target lang)

• It includes lexical and syntactic analysis, symbol table management, semantic analysis and the generation of intermediate code.

• **Code optimization** can also be done by the front end. It also includes error handling at the phases concerned.

Back End

• Dependent on the target machine and independent on the source language.

• This includes code optimization, code generation.

PASSES

- The phases of compiler can be implemented in a single pass by marking the primary actions viz. reading of input file and writing to the output file.

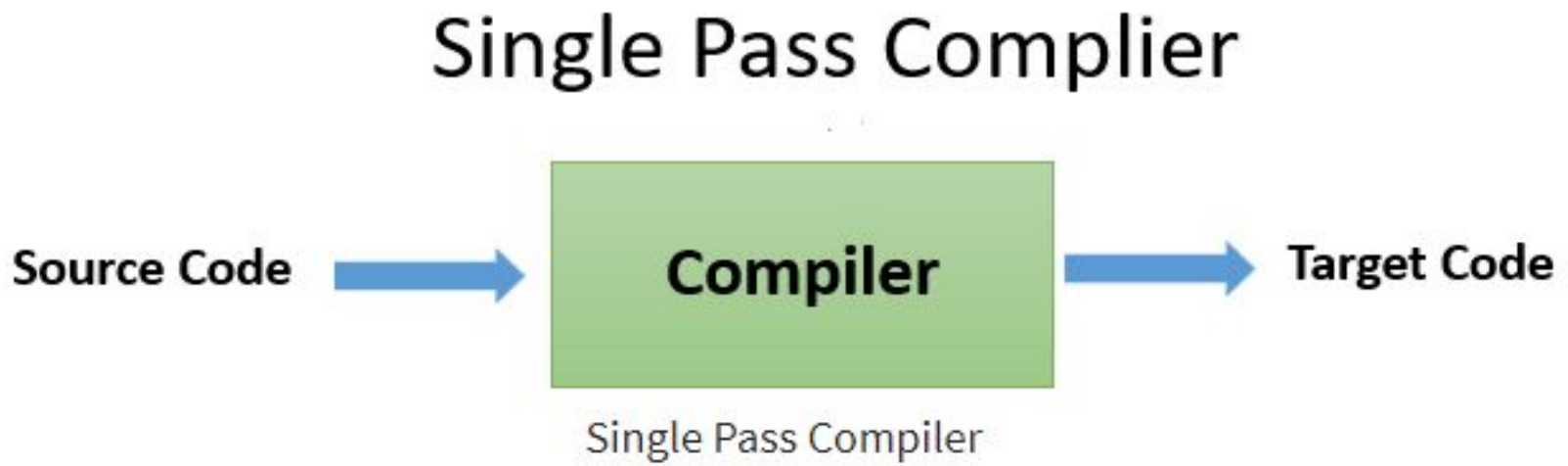
- Several phases of compiler are grouped into one pass in such a way that the operations in each and every phase are incorporated during the pass.

- Lexical analysis, syntax analysis, semantic analysis and intermediate code generation might be grouped into one pass. If so, the token stream after lexical analysis may be translated directly into intermediate code.

Types of Compiler

- Single Pass Compilers
- Two Pass Compilers
- Multi pass Compilers

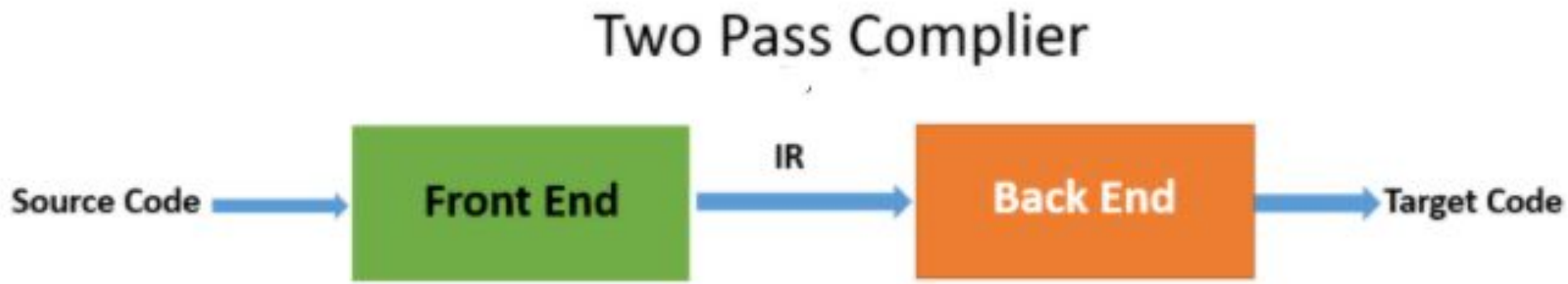
Single Pass Compiler



Single Pass Compiler

In single pass Compiler source code directly transforms into machine code. For example, Pascal language.

Two Pass Compiler



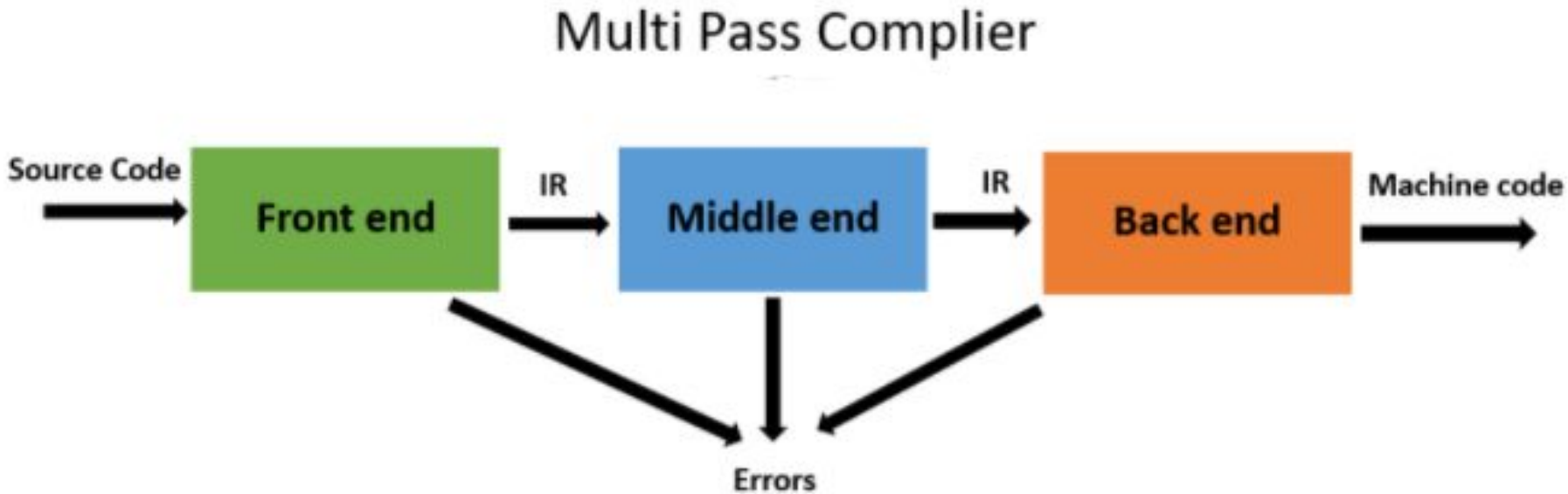
Two Pass Compiler

Two pass Compiler is divided into two sections, viz.

1. **Front end:** It maps legal code into Intermediate Representation (IR).
2. **Back end:** It maps IR onto the target machine

The Two pass compiler method also simplifies the retargeting process. It also allows multiple front ends.

Multipass Compilers



Multipass Compilers

The multipass compiler processes the source code or syntax tree of a program several times. It divided a large program into multiple small programs and process them. It develops multiple intermediate codes. All of these multipass take the output of the previous phase as an input. So it requires less memory. It is also known as 'Wide Compiler'.

Features of Compilers

- Correctness
- Speed of compilation
- Preserve the correct the meaning of the code
- The speed of the target code
- Recognize legal and illegal program constructs
- Good error reporting/handling
- Code debugging help

Task of the Compiler

- Breaks up the up the source program into pieces and impose grammatical structure on them
- Allows you to construct the desired target program from the intermediate representation and also create the symbol table
- Compiles source code and detects errors in it
- Manage storage of all variables and codes.
- Support for separate compilation
- Read, analyze the entire program, and translate to semantically equivalent
- Translating the source code into object code depending upon the type of machine

Compiler Construction Tool

Compiler construction tools

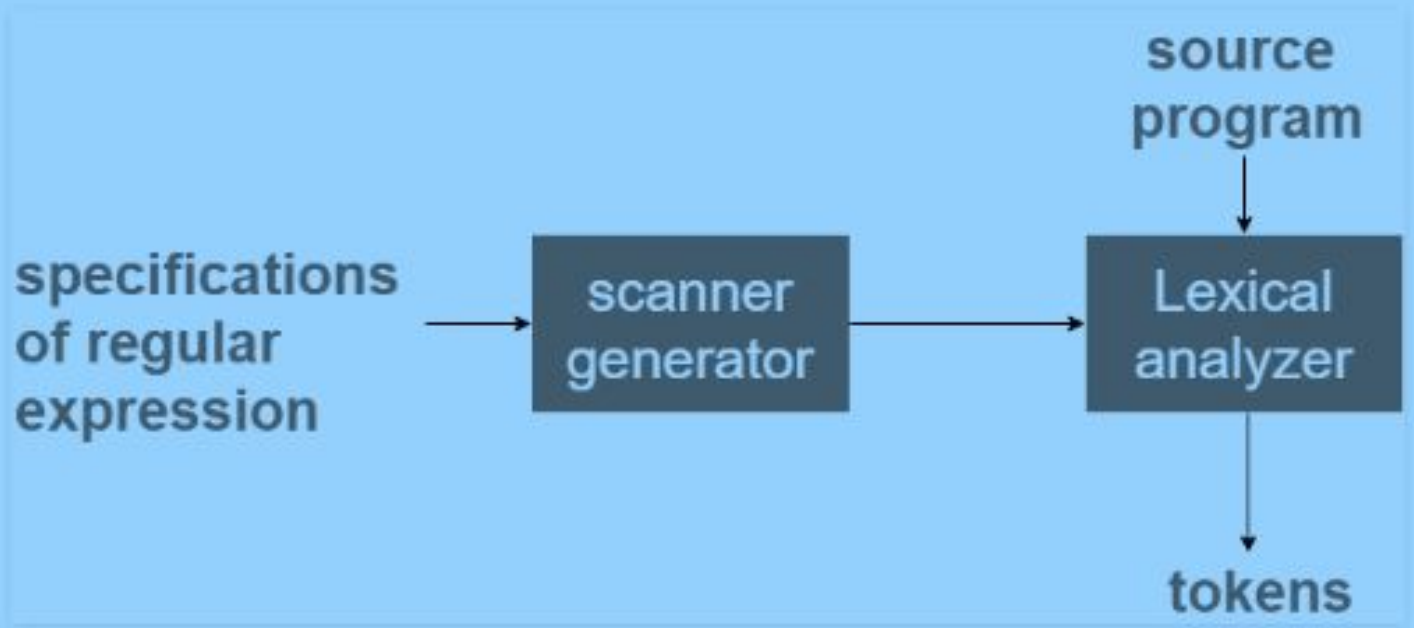
The compiler writer can use some specialized tools that help in implementing various phases of a compiler. These tools assist in the creation of an entire compiler or its parts. Some commonly used compiler construction tools include:

1. **Scanner Generator**
2. **Parser Generator**
3. **Syntax –Directed Translation engine**
4. **Automatic code generators**
5. **Data-flow analysis engines**
6. **Compiler-Construction toolkits**

1 **Scanner Generator –**

It generates lexical analyzers from the input that consists of regular expression description based on tokens of a language. It generates a finite automation to recognize the regular expression.

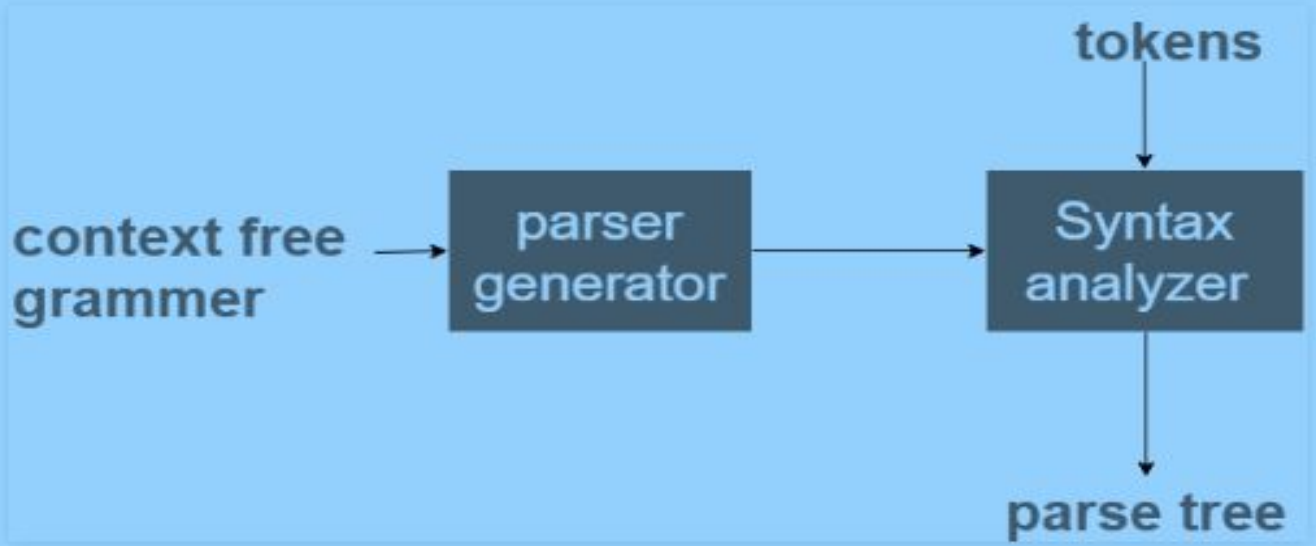
Example: Lex



2 **Parser Generator –**

It produces syntax analyzers (parsers) from the input that is based on a grammatical description of programming language or on a context-free grammar. It is useful as the syntax analysis phase is highly complex and consumes more manual and compilation time.

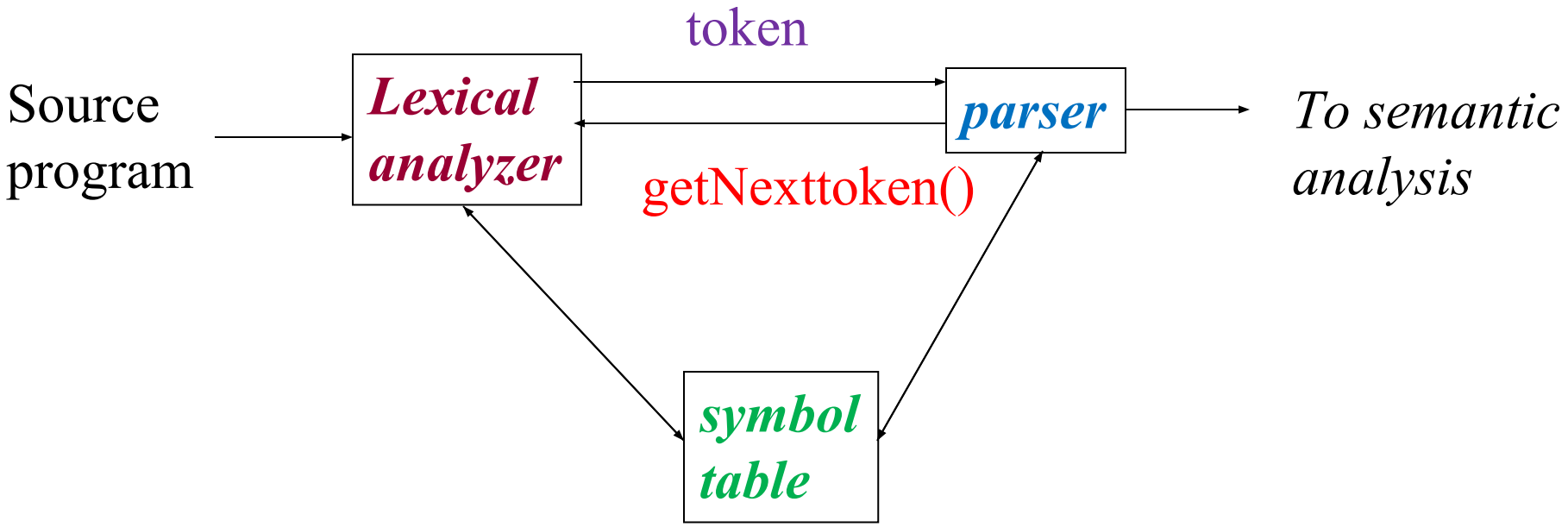
Example:PIC, EQM



3. **Syntax directed translation engines –**
   It generates intermediate code with three address format from the input that consists of a parse tree. These engines have routines to traverse the parse tree and then produces the intermediate code. In this, each node of the parse tree is associated with one or more translations.
4. **Automatic code generators –**
   It generates the machine language for a target machine. Each operation of the intermediate language is translated using a collection of rules and then is taken as an input by the code generator. Template matching process is used. An intermediate language statement is replaced by its equivalent machine language statement using templates.
5. **Data-flow analysis engines –**
   It is used in code optimization.Data flow analysis is a key part of the code optimization that gathers the information, that is the values that flow from one part of a program to another
6. **Compiler construction toolkits –**
   It provides an integrated set of routines that aids in building compiler components or in the construction of various phases of compiler.

Role Of Lexical Analyzer

Interaction of Lexical analyzer with parser



Tasks –lexical analyzer

- **Separation of the input source code into tokens**.
- **Stripping out the unnecessary white spaces** from the source code.
- **Removing the comments** from the source text.
- **Keeping track of line numbers** while scanning the new line characters. These line numbers are used by the error handler to print the error messages.
- **Preprocessing of macros.**

Issues in Lexical Analysis

- **There are several reasons for separating the analysis phase of compiling into lexical analysis and parsing:**
- It leads to ***simpler design*** **of the parser** as the unnecessary tokens can be eliminated by scanner.
- **Efficiency** of the process of compilation is improved. The lexical analysis phase is most time consuming phase in compilation. Using ***specialized buffering*** to improve the speed of compilation.
- **Portability** of the compiler is enhanced as the specialized symbols and characters(language and machine specific) are isolated during this phase.

Tokens, Patterns, Lexemes

- Connected with lexical analysis are three important terms with similar meaning.
- Lexeme
- Token
- Patterns

Tokens, Patterns, Lexemes

- A ***token*** is a pair consisting of a **token name and an optional attribute value**. Token name: Keywords, operators, identifiers, constants, literal strings, punctuation symbols(such as commas,semicolons)
- A ***lexeme*** is a sequence of characters in the source program that matches the pattern for a token and is identified by the lexical analyzer as an **instance of that token**. E.g.**Relation** {<.<=,>,>=,==,<>}

- A ***pattern*** is a ***description of the form that the lexemes of token may take***.
- It gives an **informal or formal description of a token.**
- ***Eg: identifier***
- **2 purposes**
- Gives a precise description/ specification of tokens.
- Used to automatically generate a lexical analyzer

Example

Token	Informal description	Sample lexemes
if	Characters i, f	if
else	Characters e, l, s, e	else
comparison	< or > or <= or >= or == or !=	<=, !=
id	Letter followed by letter and digits	pi, score, D2
number	Any numeric constant	3.14159, 0, 6.02e23
literal	Anything but " sorrounded by "	"core dumped"

printf("total = %d\n", score);

Attributes for Token

Example 3.2: The token names and associated attribute values for the Fortran statement

```
E = M * C ** 2
```

are written below as a sequence of pairs.

<id, pointer to symbol-table entry for E>
<assign_op>
<id, pointer to symbol-table entry for M>
<mult_op>
<id, pointer to symbol-table entry for C>
<exp_op>
<number, integer value 2>

Lexical Analysis

- Lexical analyzer: reads input characters and produces a sequence of tokens as output (nexttoken()).
  - Trying to understand each element in a program.
  - *Token*: a group of characters having a collective meaning.

    const pi = 3.14159;

    Token 1: (const, -)
    Token 2: (identifier, ‘pi’)
    Token 3: (=, -)
    Token 4: (realnumber, 3.14159)
    Token 5: (;, -)

Lexical errors

- **1.)** let us consider a statement **"fi(a==f)".** Here "fi" is a **misspelled keyword**. This error **is not detected in lexical analysis** as **"fi" is taken as an identifier.** This error is then detected in other phases of compilation.
- **2.)** in case the lexical analyzer is not able to continue with the process of compilation, it resorts to **panic mode of error recovery**.

- ***Deleting the successive characters*** from the remaining input until a token is detected.
- ***Deleting extraneous characters.***

Lexical errors

- Some errors are out of power of lexical analyzer to recognize:
  - fi (a == f(x)) ...
- However it may be able to recognize errors like:
  - d = 2r
- Such errors are recognized when no pattern for tokens matches a character sequence

Error recovery

- Panic mode: successive characters are ignored until we reach to a well formed token
- Delete one character from the remaining input
- Insert a missing character into the remaining input
- Replace a character by another character
- Transpose two adjacent characters

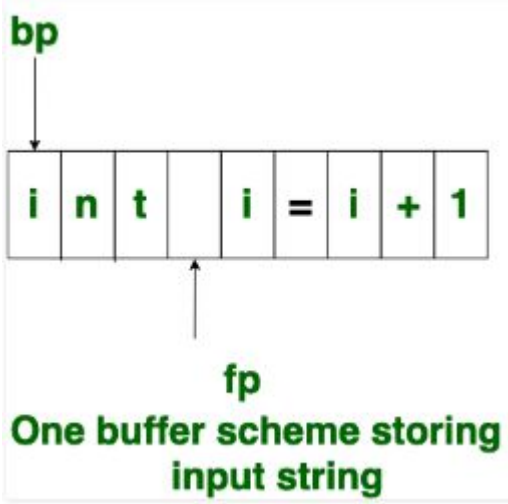
Minimum distance error correction

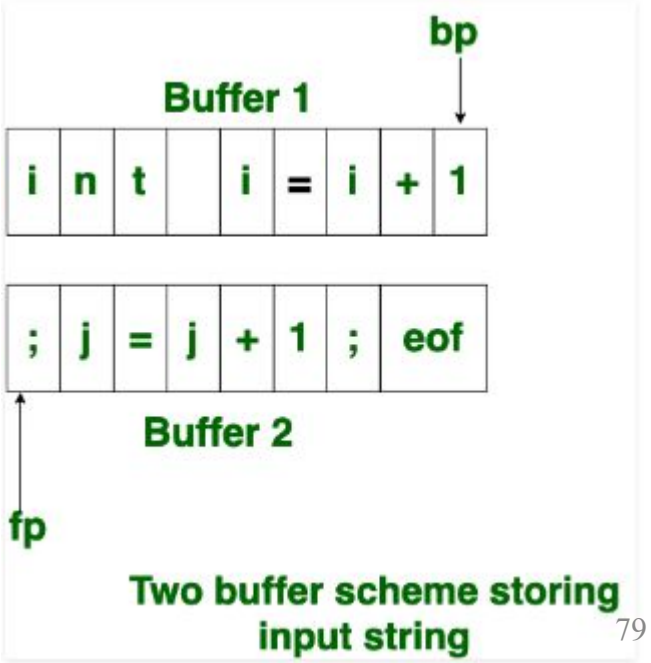
- Is the strategy generally followed by the **lexical analyzer to correct the errors in the lexemes.**
- It is nothing but the **minimum number of the corrections to be made to convert an invalid lexeme to a valid lexeme**.
- But it is ***not generally used in practice*** because it is ***too costly*** to implement.

Buffering Techniques

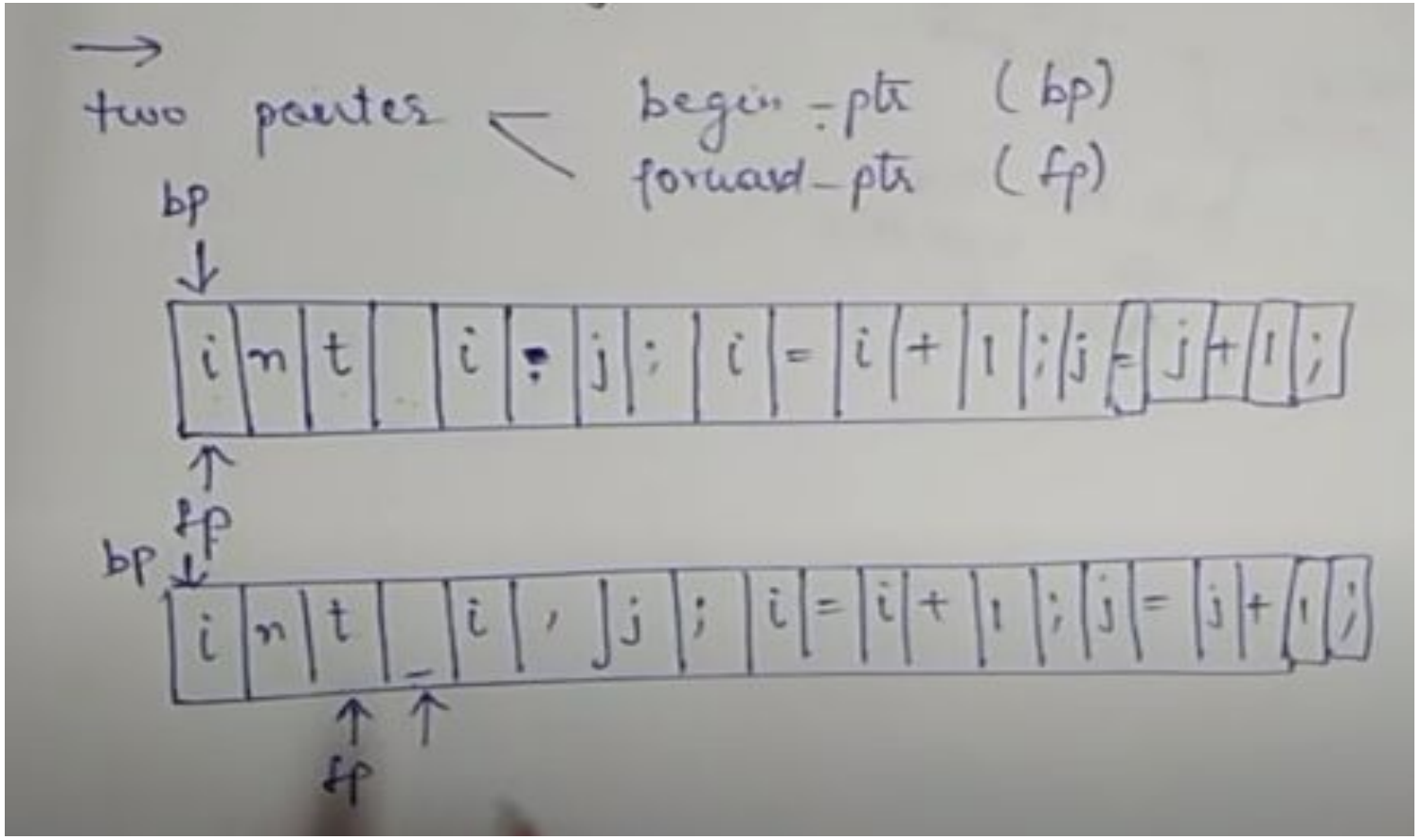
Input Buffering

- How to increase the speed of reading the source program, character by character ?
- A specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.
- There are two type of buffering scheme – One buffer scheme and two buffer scheme

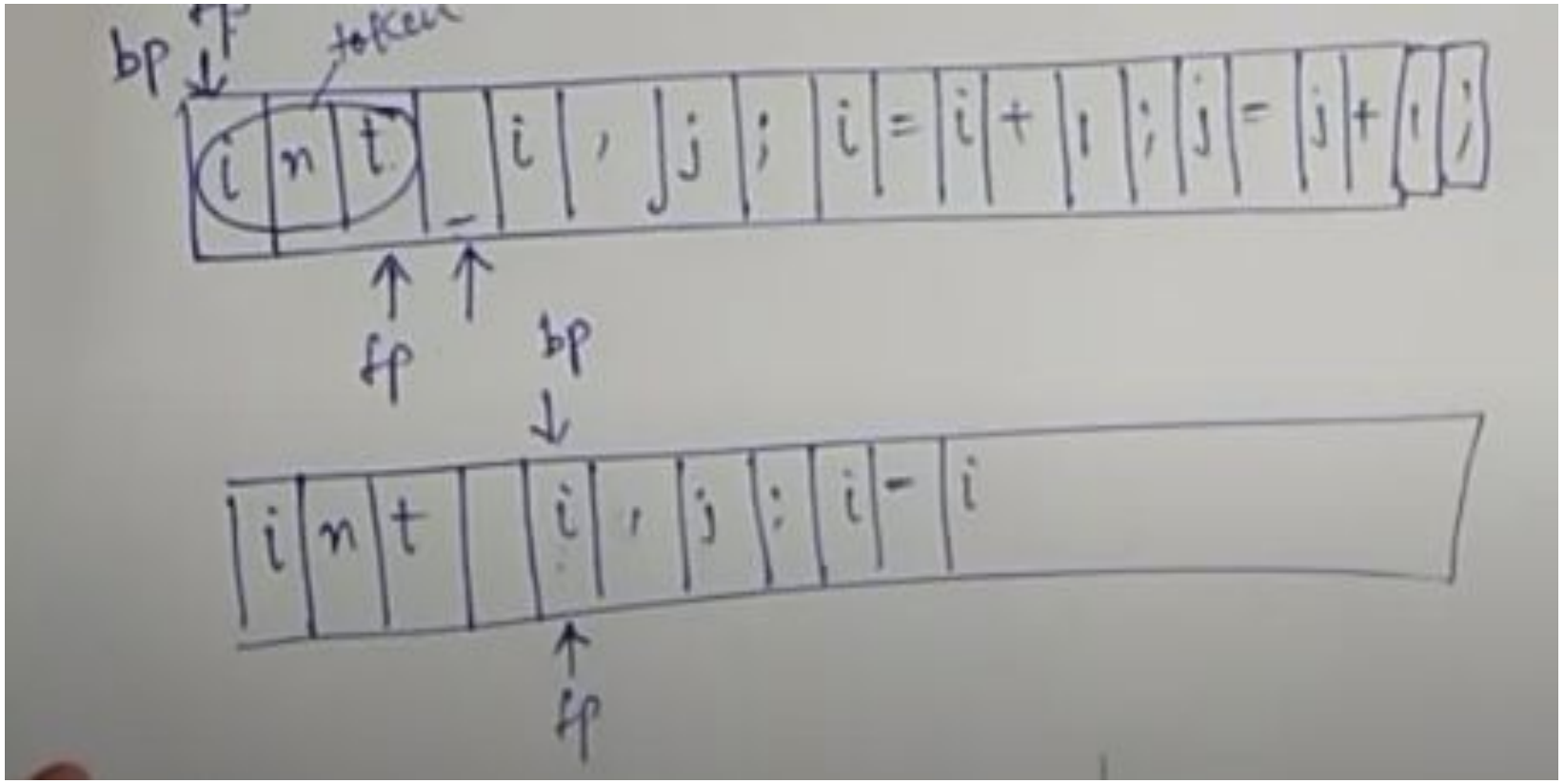




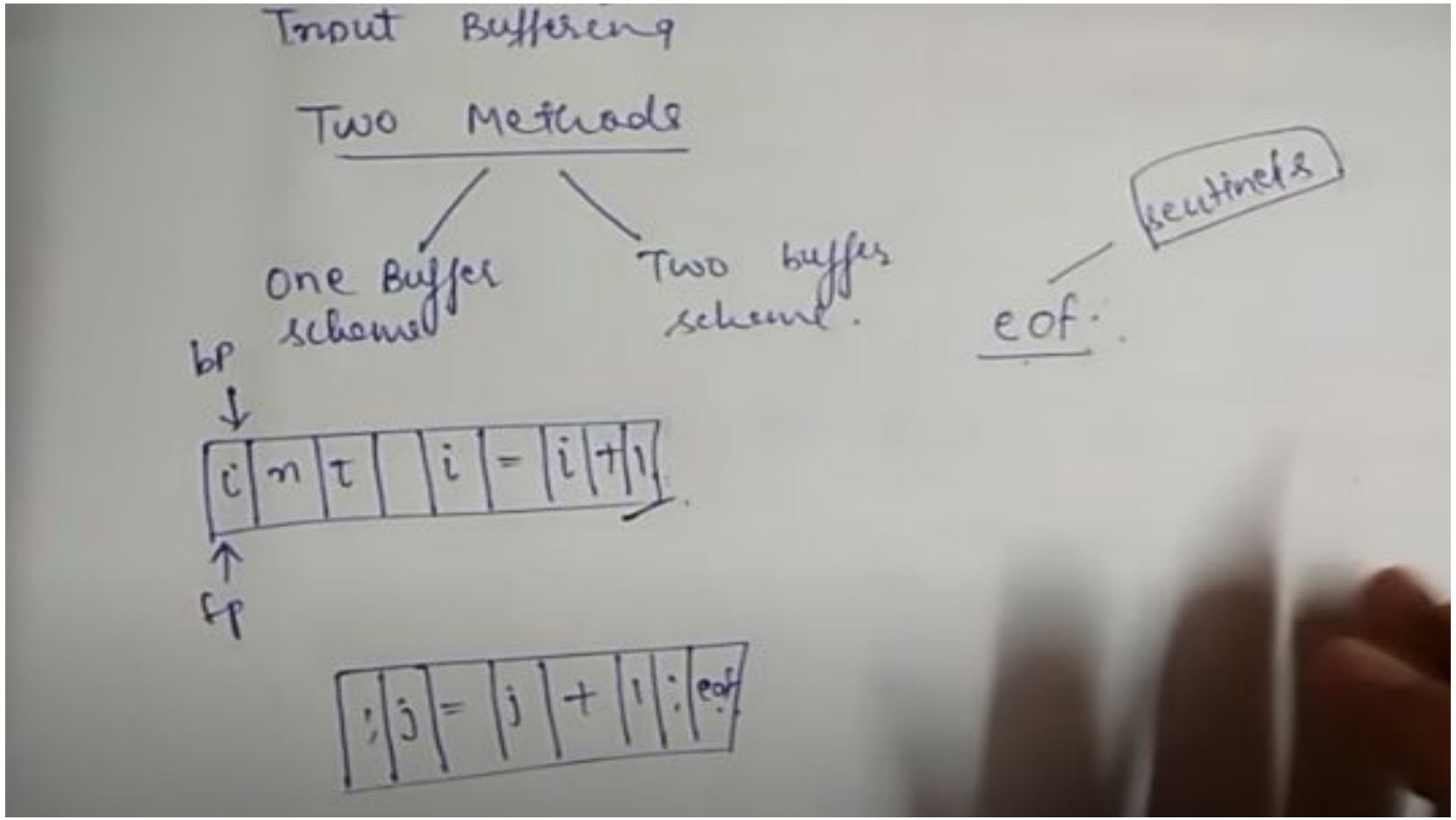
Cont..,



Cont..,



Cont..,


Input Buffering
Two Methods
One Buffer scheme
Two buffer scheme.
sentinels
eof.
bp
fp
i n t i = i + 1
; j = j + 1 ; eof

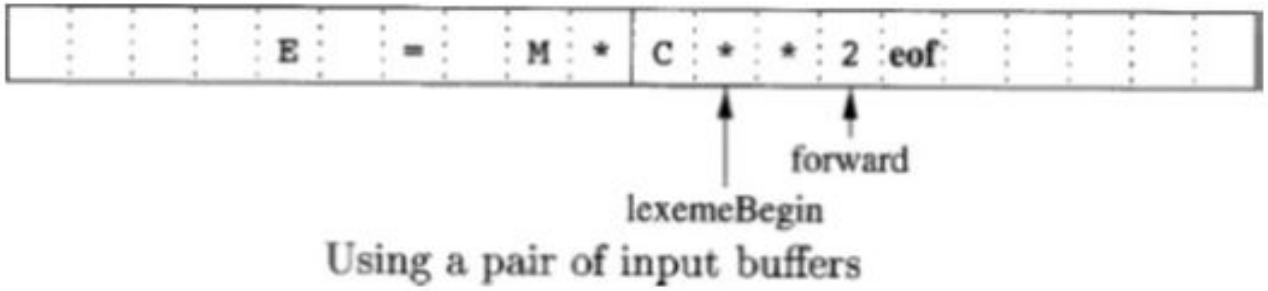
Input Buffering

- Speedup the reading the source program
- Look one or more characters beyond the next lexeme
- There are many situations where we need to look at least one additional character ahead.

Main Concept of Input Buffering

- Two pointers to the input are maintained
  - **Pointer lexemeBegin**, marks the beginning of the current lexeme, whose extent we are attempting to determine.
  - **Pointer forward** scans ahead until a pattern match is found;
- Sentinels
  - To reload the buffer and to advance the forward pointer effectively we have to make two tests
  - one for the end of the buffer and other to determine what character is read
  - This can be done in a combine way if we hold a sentinel character at the end
  - ***A sentinel is a special character that cannot be part of the source program (Eg: eof)***

Buffer Pairs



Using a pair of input buffers

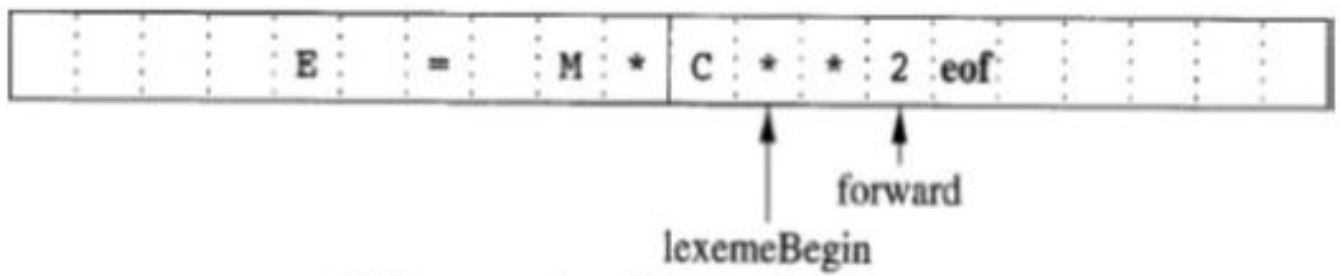
- Each buffer is of the same size N, and N is usually the size of a disk block, e.g., 4096 bytes.
- Using one system read command we can read N characters in a buffer, rather than using one system call per character.
- If fewer than N characters remain in the input file, then a

Input Buffering

- For instance, we cannot be sure we've seen the end of an identifier until we see a character that is not a letter or digit, and therefore is not part of the lexeme for id.
- In C, single-character operators like -, =, or < could also be the beginning of a two-character operator like ->, ==, or <=.
- A a two-buffer scheme that handles large lookaheads safely.
- We then consider an improvement involving "sentinels" that saves time checking for the ends of buffers.

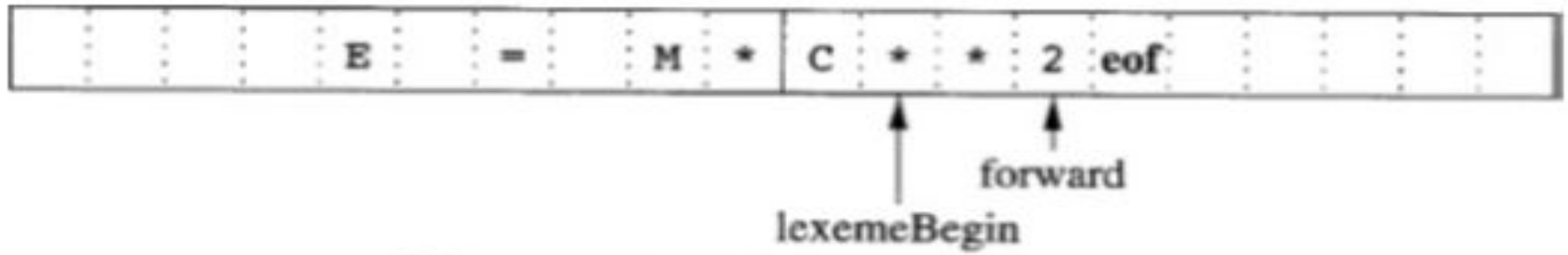
Buffer Pairs

- The amount of time taken is high to process characters of a large source program.
- Specialized buffering techniques have been developed to reduce the amount of overhead required to process a single input character.
- An important scheme involves two buffers that are alternately reloaded, as suggested in figure.



Using a pair of input buffers

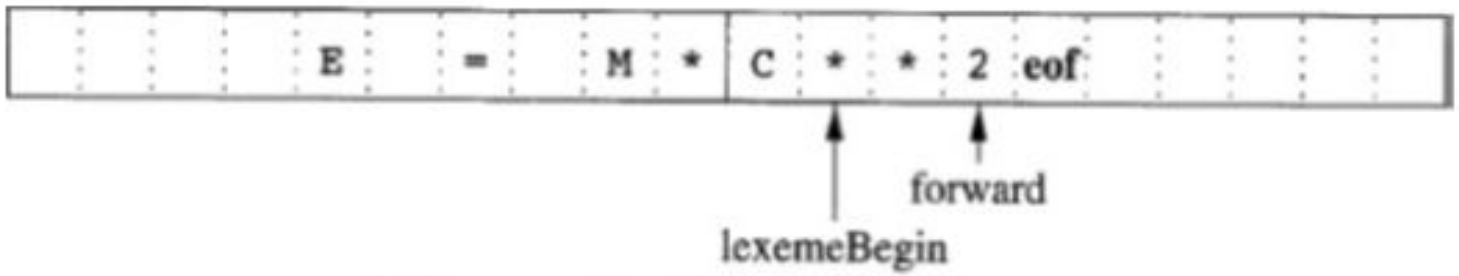
Buffer Pairs



Using a pair of input buffers

- If fewer than N characters remain in the input file, then a special character, represented by **eof**, marks the end of the source file.
- This **eof** is different from any possible character of the source program.

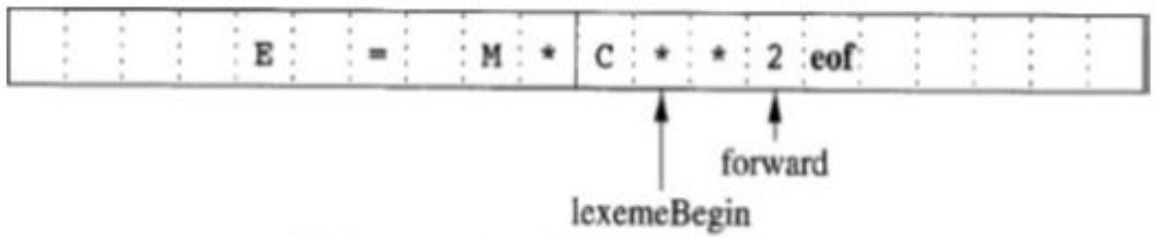
Buffer Pairs



Using a pair of input buffers

- Two pointers to the input are maintained:
  - Pointer *lexemeBegin*, marks the beginning of the current lexeme, whose extent we are attempting to determine.
  - Pointer *forward* scans ahead until a pattern match is found.

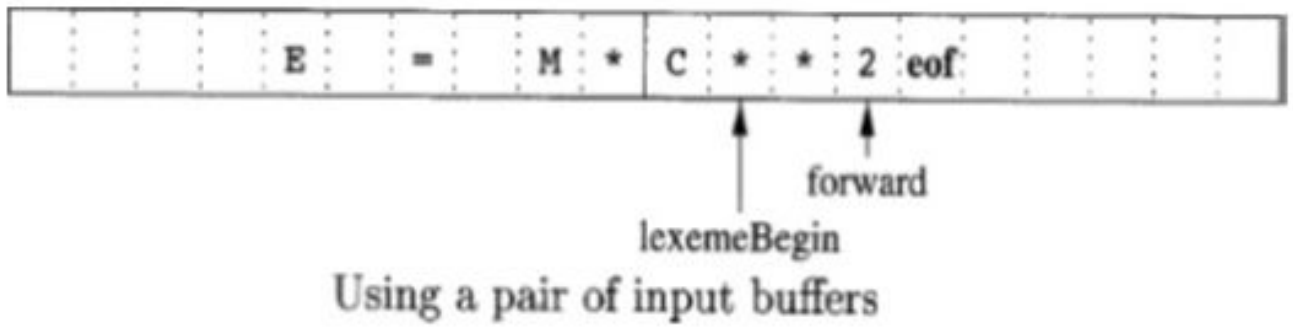
Buffer Pairs



Using a pair of input buffers

- Advancing *forward* requires that we first test whether we have reached the end of one of the buffers.
- If so, we must reload the other buffer from the input. And move *forward* to the beginning of the newly loaded buffer.

Sentinels



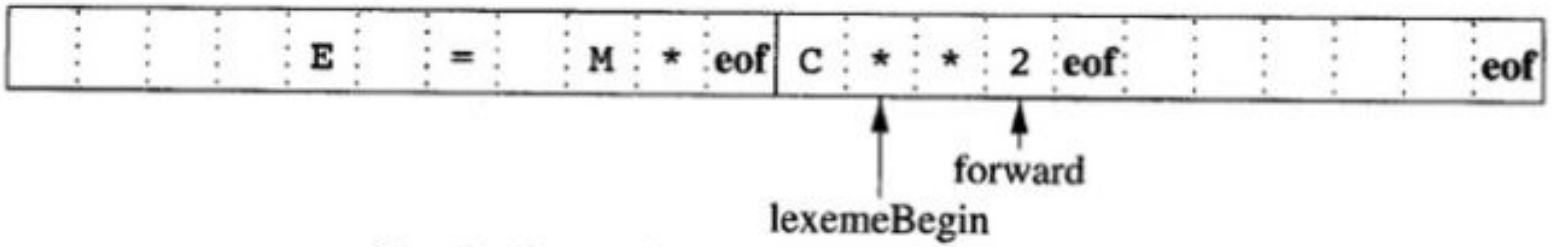
Using a pair of input buffers

- If we use the previous scheme as described, we must check, each time we advance forward, that we have not moved off one of the buffers.
- If we do, then we must also reload the other buffer.
- Thus, for each character read, we make two tests:
  - one for the end of the buffer.
  - And one to determine what character is read (the latter may be a multiway branch).

Sentinels

- Two tests can be simplified using additional sentinels
- We can combine the buffer-end test with the test for the current character if we extend each buffer to hold a sentinel character at the end.
- The sentinel is a special character that cannot be part of the source program, and a natural choice is the character **eof**.

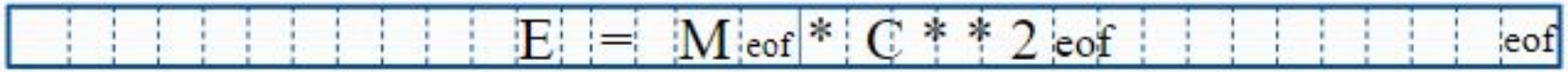
Sentinels



Sentinels at the end of each buffer

- Figure shows the same arrangement as previous, but with the sentinels added.
- Note that **eof** retains its use as a marker for the end of the entire input.
- Any **eof** that appears other than at the end of a buffer means that the input is at an end.

Sentinels




```
Switch (*forward++) {
    case eof:
            if (forward is at end of first buffer) {
                    reload second buffer;
                    forward = beginning of second buffer;
            }
            else if {forward is at end of second buffer) {
                    reload first buffer;\
                    forward = beginning of first buffer;
            }
            else /* eof within a buffer marks the end of input */
                    terminate lexical analysis;
            break;
    cases for the other characters;
}
```

Conclusion

Buffer Pairs

- Lexical analyzer needs to look ahead several characters beyond the lexeme for a pattern before a match can be announced.
- Use a function **ungetc** to push look-ahead characters back into the input stream.
- Large amount of time can be consumed moving characters.

Special Buffering Technique

Use a buffer divided into two N-character halves

N = Number of characters on one disk block

One system command read N characters

Fewer than N character => eof

The Lexical-Analyzer Generator Lex

- Lex is a tool in lexical analysis phase to recognize tokens using regular expression.
- Lex tool itself is a lex compiler.

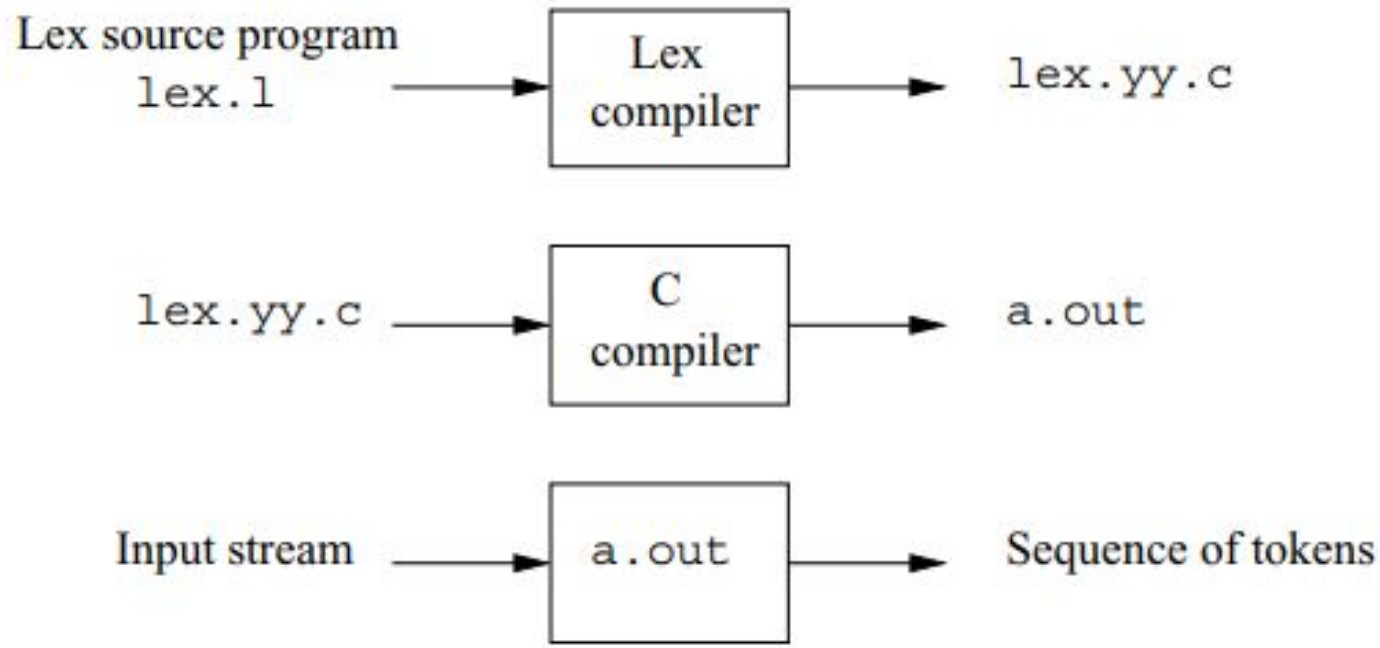


Figure 3.22: Creating a lexical analyzer with Lex

- ***lex.l*** is an a input file written in a language which describes the generation of lexical analyzer. The lex compiler transforms *lex.l* to a C program known as *lex.yy.c.*

- ***lex.yy.c*** is compiled by the C compiler to a file called *a.out.*

- The output of C compiler is the working lexical analyzer which takes stream of input characters and produces a stream of tokens.

- ***yylval*** is a global variable which is shared by lexical analyzer and parser to return the name and an attribute value of token.

- The attribute value can be numeric code, pointer to symbol table or nothing.

- Another tool for lexical analyzer generation is Flex

Structure of Lex Programs

Lex program will be in following form

A Lex program has the following form:

declarations
%%
translation rules
%%
auxiliary functions

```
%{
    /* definitions of manifest constants
    LT, LE, EQ, NE, GT, GE,
    IF, THEN, ELSE, ID, NUMBER, RELOP */
%}

/* regular definitions */
delim     [ \t\n]
ws        {delim}+
letter    [A-Za-z]
digit     [0-9]
id        {letter}({letter}|{digit})*
number    {digit}+(\.{digit}+)?(E[+-]?{digit}+)?

%%

{ws}      {/* no action and no return */}
if        {return(IF);}
then      {return(THEN);}
else      {return(ELSE);}
```

```
{id}      {yylval = (int) installID(); return(ID);}
{number}  {yylval = (int) installNum(); return(NUMBER);}
"<"       {yylval = LT; return(RELOP);}
"<="      {yylval = LE; return(RELOP);}
"="       {yylval = EQ; return(RELOP);}
"<>"      {yylval = NE; return(RELOP);}
">"       {yylval = GT; return(RELOP);}
">="      {yylval = GE; return(RELOP);}

%%

int installID() {/* function to install the lexeme, whose
                    first character is pointed to by yytext,
                    and whose length is yyleng, into the
                    symbol table and return a pointer
                    thereto */
}
```

Declarations This section includes declaration of variables, constants and regular definitions.

Translation rules It contains regular expressions and code segments.

Form : Pattern {Action}

Pattern is a regular expression or regular definition.

Action refers to segments of code.

- ***Auxiliary functions*** This section holds additional functions which are used in actions. These functions are compiled separately and loaded with lexical analyzer.
- Lexical analyzer produced by lex starts its process by reading one character at a time until a valid match for a pattern is found.
- Once a match is found, the associated action takes place to produce token.
- The token is then given to parser for further processing.

Conflict Resolution in Lex

- Conflict arises when several prefixes of input matches one or more patterns. This can be resolved by the following:
- Always prefer a longer prefix than a shorter prefix.
- If two or more patterns are matched for the longest prefix, then the first pattern listed in lex program is preferred.

Lookahead Operator

- Lookahead operator is the additional operator that is read by lex in order to distinguish additional pattern for a token.
- Lexical analyzer is used to read one character ahead of valid lexeme and then retracts to produce token.
- At times, it is needed to have certain characters at the end of input to match with a pattern. In such cases, slash (/) is used to indicate end of part of pattern that matches the lexeme.

❖ (eg.) In some languages keywords are not reserved. So the statements

❖ IF (I, J) = 5 and IF(condition) THEN results in conflict whether to produce IF as an array name or a keyword. To resolve this the lex rule for keyword IF can be written as,

❖ IF/\ (.* \) { letter }

Issues in Lexical Analysis

1. Lookahead
2.Ambiguity

SPECIFICATION OF TOKENS USING REGULAR EXPRESSION

Specification of tokens

- **Scanners** are special **pattern matching processors.**
- For representing patterns of strings of characters, **Regular Expressions(RE**) are used.
- *A regular expression (r) is defined by set of strings that matches it.*
- This set is called as the **language** generated by the regular expression and is represented as

- An alphabet is a finite set of symbols.
- Example
- A set of alphabetic characters is represented as L={A,…,Z,a,…,z} and set of digits is represented as D={0,1,…,9}.
- LUD is a language.
- *Strings over LUD*- Begin,Max1, max1, 123, €…

Operations on Languages

OPERATION	DEFINITION
union of L and M written $L \cup M$	$L \cup M = \{ s \mid s \text{ is in } L \text{ or } s \text{ is in } M \}$
concatenation of L and M written LM	$LM = \{ st \mid s \text{ is in } L \text{ and } t \text{ is in } M \}$
Kleene closure of L written L^*	$L^* = \bigcup_{i=0}^{\infty} L^i$ L^* denotes "zero or more concatenations of" L.

- **Intersection**
- $L \cap M = \{ s \mid s \text{ is in } L \text{ and } S \text{ is in } M \}$
- **Exponentiation**
- $L^{i} = L\, L^{i-1}$

Regular expression operations

- Choice among alternates
- Concatenation
- Repetition

1. CHOICE AMONG ALTERNATES

- Indicated by metacharacter **'|'(vertical bar)**
- **r|s**
- R.E that matches any string that is matched **either by r or s.**
- **L(r|s)= L(r) U L(s)**

2. CONCATENATION

- **rs**
- It matches any string that is a concatenation of 2 strings, **the first of which matches r and second of which matches s.**
- **L(rs) = L(r) L(s)**

3. REPETITION

- Also called **Kleene closure**
- Represents **any finite concatenation of strings each matches strings from L(r).**
- **r***
- Let S={a}, then L(a*)={€, a, aa, aaa,…}
- S*={€}USUSSUSSSU….=
- 

Precedence of operators

- Repetition ----------(highest)
- Concatenation
- Choice----------(lowest)

left associative

- One or more instances: **(r)+**
- Zero of one instances: **r?**
- Zero or more instances: **r***

Specification of tokens

- In theory of compilation regular expressions are used to formalize the specification of tokens
- Regular expressions are means for specifying regular languages
- Example:
  - Letter_(letter_ | digit)*
- Each regular expression is a pattern specifying the form of strings

Regular expressions

SRM INSTITUTE OF SCIENCE & TECHNOLOGY (Deemed to be University u/s 3 of UGC Act, 1956)

- ε is a regular expression, L(ε) = {ε}
- If a is a symbol in Σ then a is a regular expression, L(a) = {a}
- (r) | (s) is a regular expression denoting the language L(r) $\cup$ L(s)
- (r)(s) is a regular expression denoting the language L(r)L(s)
- (r)* is a regular expression denoting (L(r))*
- (r) is a regular expression denting L(r)

Regular Expressions

- Regular expressions are a notation to represent lexeme pattern for a token.
- We use regular expressions to describe tokens of a programming language.
- A regular expression is built up of simpler regular expressions (using defining rules)
- Each regular expression denotes a language.
- A language denoted by a regular expression is called as a **regular set**.

Regular Expressions (Rules)

Regular expressions over alphabet Σ

Reg. Expr	Language it denotes
ε	{ε}
$a \in \Sigma$	{a}
$(r_1) \mid (r_2)$	$L(r_1) \cup L(r_2)$
$(r_1)\ (r_2)$	$L(r_1)\ L(r_2)$
$(r)^*$	$(L(r))^*$
(r)	L(r)

- $(r)^+ = (r)(r)^*$
- $(r)? = (r) \mid \varepsilon$

Regular Expressions (cont.)

- We may remove parentheses by using precedence rules.
  - * highest
  - concatenation next
  - | lowest
- $ab^*|c$ means $(a(b)^*)|(c)$

- Ex:
  - $\Sigma = \{0,1\}$
  - $0|1 \Rightarrow \{0,1\}$
  - $(0|1)(0|1) \Rightarrow \{00,01,10,11\}$
  - $0^* \Rightarrow \{\varepsilon,0,00,000,0000,....\}$
  - $(0|1)^*$ => all strings with 0 and 1, including the empty string

Regular Definitions

- To write regular expression for some languages can be difficult, because their regular expressions can be quite complex. In those cases, we may use *regular definitions*.
- We can give names to regular expressions, and we can use these names as symbols to define other regular expressions.

- A ***regular definition*** is a sequence of the definitions of the form:

$d_1 \rightarrow r_1$ where d_i is a distinct name and

$d_2 \rightarrow r_2$ r_i is a regular expression over symbols in $\Sigma \cup \{d_1, d_2, ..., d_{i-1}\}$

$d_n \rightarrow r_n$

Regular definitions

d1 -> r1

d2 -> r2

…

dn -> rn

If we try to write the regular expression representing identifiers without using regular definitions, that regular expression will be complex.

(A|...|Z|a|...|z) ((A|...|Z|a|...|z) | (0|...|9)) *

- Example:

letter_ -> A | B | … | Z | a | b | … | Z | _

digit -> 0 | 1 | … | 9

id -> letter_ (letter_ | digit)*

- Ex: Unsigned numbers in Pascal

digit → 0 | 1 | ... | 9

digits → digit $^{+}$

opt-fraction → (. digits) ?

opt-exponent → (E (+|-)? digits) ?

unsigned-num → digits opt-fraction opt-exponent

Extensions

- One or more instances: (r)+
- Zero or one instances: r?
- Character classes: [abc]

- Example:
  - letter_ -> [A-Za-z_]
  - digit -> [0-9]
  - id -> letter_(letter|digit)*

Recognition of tokens

- Starting point is the language grammar to understand the tokens:

stmt -> **if** expr **then** stmt

| **if** expr **then** stmt **else** stmt

| ε

expr -> term **relop** term

| term

term -> **id**

| **number**

Recognition of tokens (cont.)

- The next step is to formalize the patterns:

  digit -> [0-9]

  Digits -> digit+

  number -> digit(.digits)? (E[+-]? Digit)?

  letter -> [A-Za-z_]

  id -> letter (letter|digit)*

  If -> if

  Then -> then

  Else -> else

  Relop -> < | > | <= | >= | = | <>

- We also need to handle whitespaces:

  ws -> (blank | tab | newline)+

Token recognition

- We also want to strip whitespace, so we need definitions

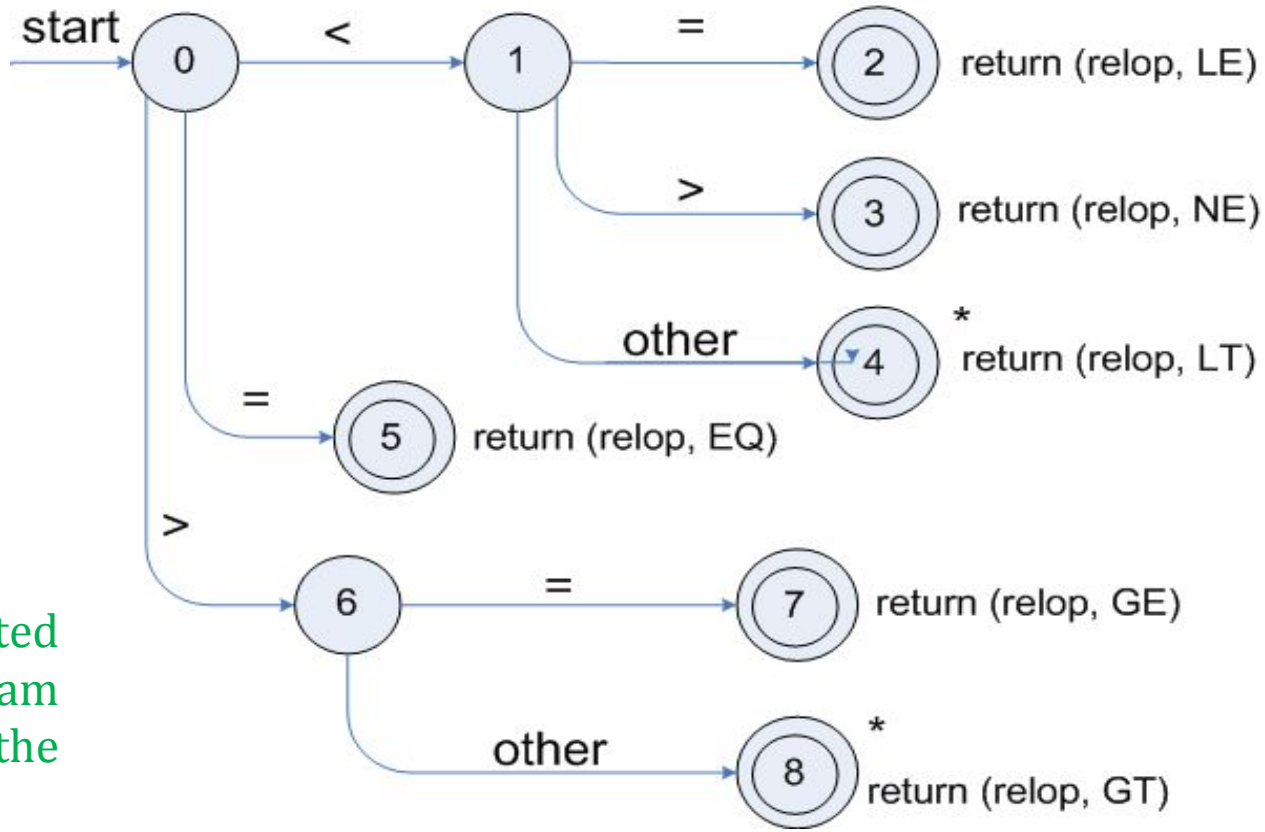
delim	->	**blank | tab | newline**
ws	->	**delim^{+}**

Operations on Languages

- Concatenation:
  - $L_1L_2 = \{ s_1s_2 \mid s_1 \in L_1 \text{ and } s_2 \in L_2 \}$
- Union
  - $L_1 \cup L_2 = \{ s \mid s \in L_1 \text{ or } s \in L_2 \}$
- Exponentiation:
  - $L^0 = \{\varepsilon\}$ $L^1 = L$ $L^2 = LL$
- Kleene Closure
  - $L^* = \bigcup_{i=0}^{\infty} L^i$
- Positive Closure
  - $L^+ = \bigcup_{i=1}^{\infty} L^i$

Transition diagrams

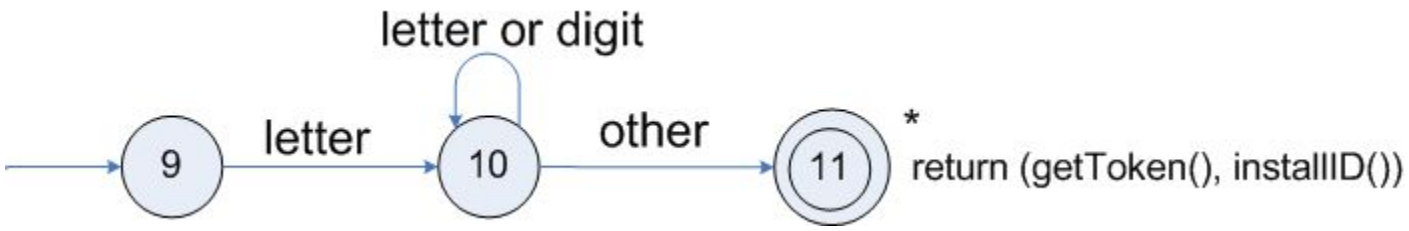
- Transition diagram for relop



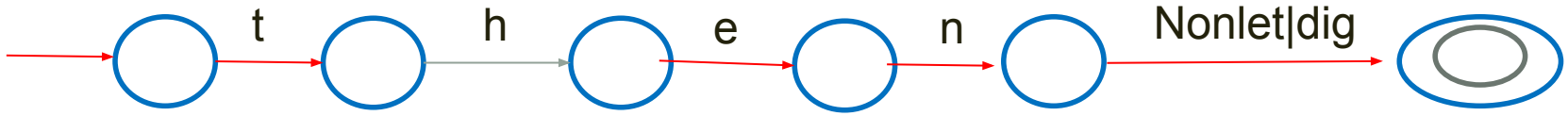
The pattern are converted into transition diagram while constructing the lexical analyzer

Transition diagrams (cont.)

- Transition diagram for reserved words and identifiers

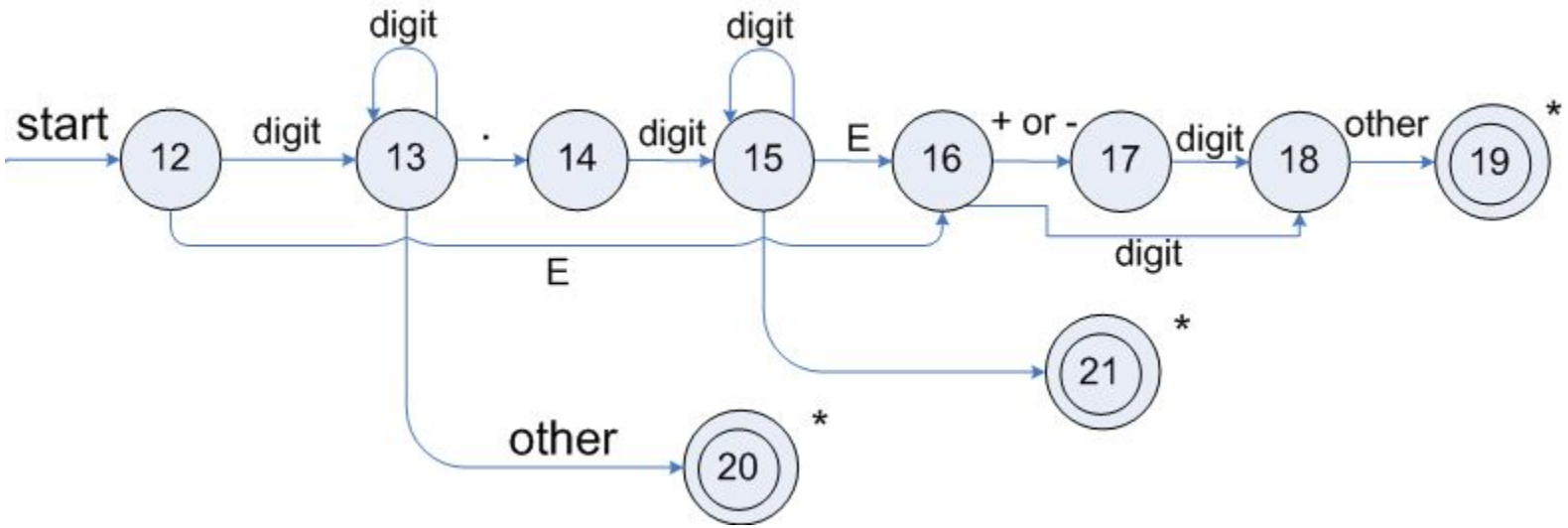


KEYWORD



Transition diagrams (cont.)

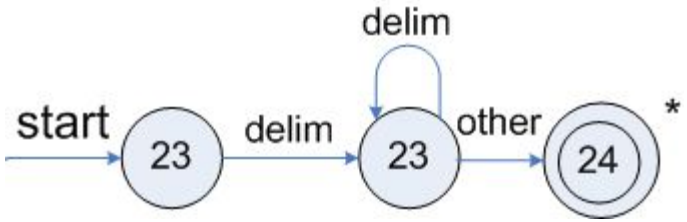
- Transition diagram for unsigned numbers



Transition diagrams (cont.)

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

- Transition diagram for whitespace



Finite Automata

- Regular expressions = specification
- Finite automata = implementation
- **Recognizer** ---A recognizer for a language is a program that takes as input a string x answers 'yes' if x is a sentence of the language and 'no' otherwise.

- A better way to convert a regular expression to a recognizer is to construct a generalized transition diagram from the expression. This diagram is called a **finite automaton.**

- Finite Automaton can be
- Deterministic
- Non-deterministic

Finite Automata

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

- A finite automaton consists of
  - An input alphabet Σ
  - A set of states S
  - A start state q0
  - A set of accepting states $F \subseteq S$
  - A set of transitions $\text{state} \rightarrow^{\text{input}} \text{state}$

Finite Automata

- Transition

$$s_1 \rightarrow^a s_2$$

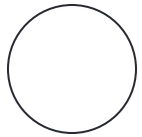
- Is read

In state s_1 on input "a" go to state s_2

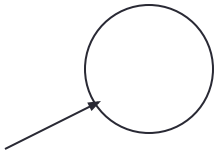
- If end of input
  - If in accepting state => accept, otherwise => reject
- If no transition possible => reject

Finite Automata State Graphs

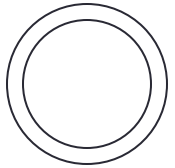
- A state



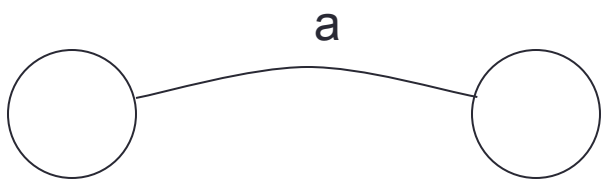
- The start state



- An accepting state



- A transition



Finite Automata

- A *recognizer* for a language is a program that takes a string x, and answers "yes" if x is a sentence of that language, and "no" otherwise.
- We call the recognizer of the tokens as a *finite automaton*.
- A finite automaton can be: *deterministic(DFA)* or *non-deterministic (NFA)*
- This means that we may use a deterministic or non-deterministic automaton as a lexical analyzer.
- Both deterministic and non-deterministic finite automaton recognize regular sets.
- Which one?
  - deterministic – faster recognizer, but it may take more space
  - non-deterministic – slower, but it may take less space
  - Deterministic automatons are widely used lexical analyzers.
- First, we define regular expressions for tokens; Then we convert them into a DFA to get a lexical analyzer for our tokens.
  - Algorithm1: Regular Expression NFA DFA (two steps: first to NFA, then to DFA)
  - Algorithm2: Regular Expression DFA (directly convert a regular expression into a DFA)

Non-Deterministic Finite Automaton (NFA)

- A non-deterministic finite automaton (NFA) is a mathematical model that consists of:
  - Q - a set of states
  - Σ - a set of input symbols (alphabet)
  - ***δ*** -move – a transition function move to map state-symbol pairs to sets of states.
  - $_{q0}$ - a start (initial) state
  - F – a set of accepting states (final states)

- ε- transitions are allowed in NFAs. In other words, we can move from one state to another one without consuming any symbol.
- A NFA accepts a string x, if and only if there is a path from the starting state to one of accepting states such that edge labels along this path spell out x.

Deterministic and Nondeterministic Automata

- Deterministic Finite Automata (DFA)
  - One transition per input per state
  - No ε-moves
- Nondeterministic Finite Automata (NFA)
  - Can have multiple transitions for one input in a given state
  - Can have ε-moves
- *Finite* automata have *finite* memory
  - Need only to encode the current state

1.14 Finite Automata

A recognizer for a language is a program that takes as input a string x and answers *yes* if x is a sentence of the language and *no* otherwise.

A regular expression is compiled into a recognizer by constructing a generalized transition diagram called a Finite Automaton (FA).

A finite automata can be Non-deterministic Finite Automata (NFA) or Deterministic Finite Automata (DFA).

It is given by $M = (Q, \Sigma, q_0, F, \delta)$.

where
- Q – Set of states
- Σ – Set of input symbols
- q_0 – Start state
- F – Set of final states
- δ – Transition function (mapping states to input symbol).

$$\delta : Q \times \Sigma \rightarrow Q$$

✓ Non-deterministic Finite Automata (NFA)

- More than one transition occurs for any input symbol from a state.
- Transition can occur even on empty string (ϵ).

- ✓ Deterministic Finite Automata (DFA)
  - ◇ For each state and for each input symbol, exactly one transition occurs from that state.

Regular expression can be converted into DFA by the following methods:

(i) Thompson's sub-set construction

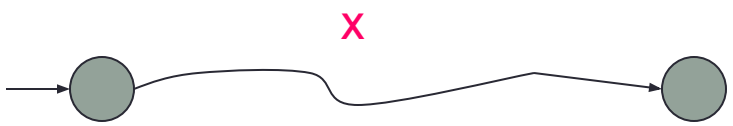
- ✓ Given regular expression is converted into NFA
- ✓ Resultant NFA is converted into DFA

(ii) Direct method

- ✓ In direct method, given regular expression is converted directly into DFA.

DFA

- For every string x, there is a unique path from initial state and associated with x.



x is accepted if and only if this path ends at a final state.

NFA

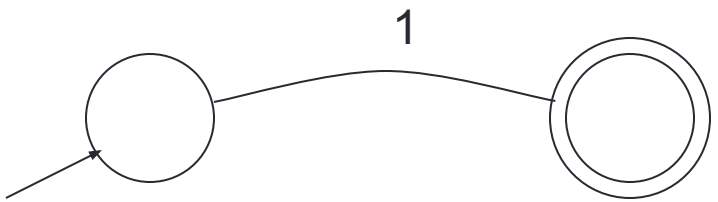
SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

- For any string x, there may exist none or more than one path from initial state and associated with x.

A Simple Example

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

- A finite automaton that accepts only “1”

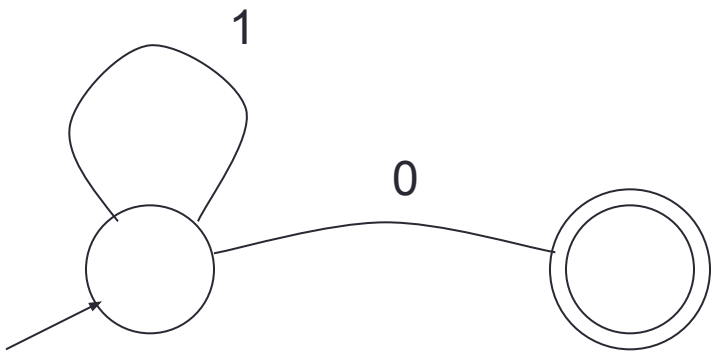


- A finite automaton accepts a string if we can follow transitions labeled with the characters in the string from the start to some accepting state

Another Simple Example

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

- A finite automaton accepting any number of 1's followed by a single 0
- Alphabet: {0,1}



- Check that "1110" is accepted.

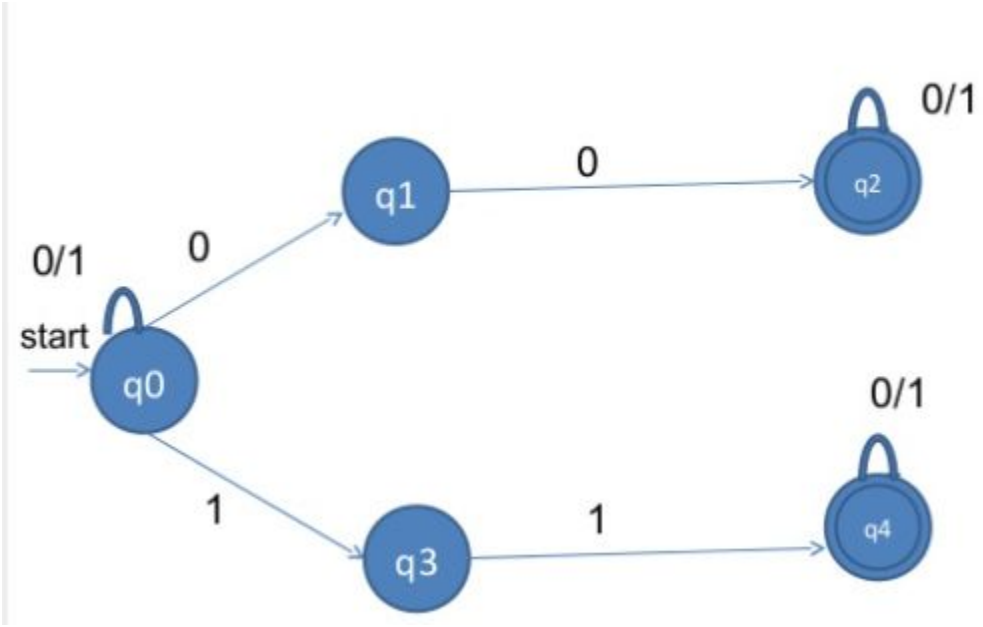
NFA

SRM
INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

Design a NFA for the language L=all strings over {0,1} that have atleast two consecutive 0's or 1's

NFA

Design a NFA for the language L=all strings over {0,1} that have atleast two consecutive 0's or 1's



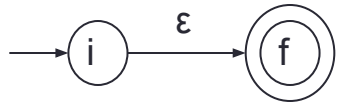
Transition Table

	0	1
→ qo	{qo,q1}	{qo,q3}
q1	q2	?
* q2	q2	q1
q3	?	q4
*q4	Q4	Q4

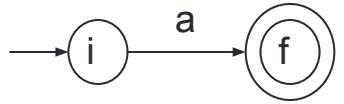
Converting A Regular Expression into A NFA (Thomson's Construction)

- This is one way to convert a regular expression into a NFA.
- There can be other ways (much efficient) for the conversion.
- Thomson's Construction is simple and systematic method. It guarantees that the resulting NFA will have exactly one final state, and one start state.
- Construction starts from simplest parts (alphabet symbols).
- To create a NFA for a complex regular expression, NFAs of its sub-expressions are combined to create its NFA,

Thomson's Construction (cont.)

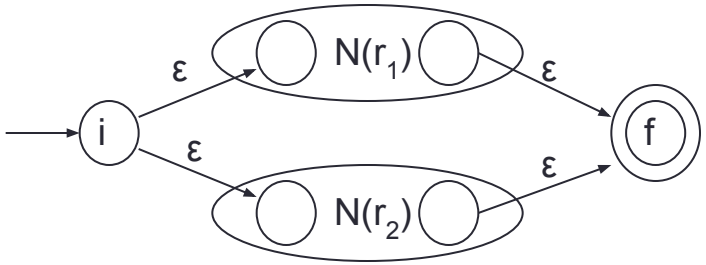


- To recognize an empty string ε



- To recognize a symbol a in the alphabet Σ

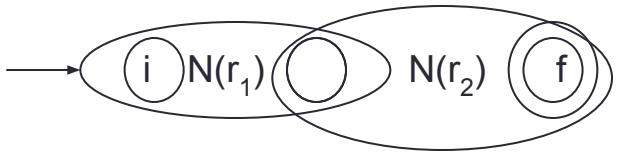
- If $N(r_1)$ and $N(r_2)$ are NFAs for regular expressions r_1 and r_2
  - For regular expression $r_1 \mid r_2$



NFA for $r_1 \mid r_2$

Thomson's Construction (cont.)

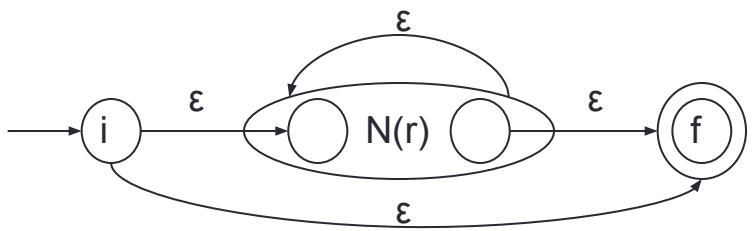
- For regular expression $r_1 r_2$



NFA for $r_1 r_2$

Final state of $N(r_2)$ become final state of $N(r_1 r_2)$

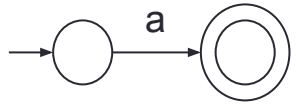
- For regular expression r^*



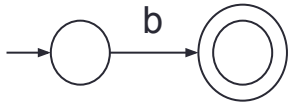
NFA for r^*

Thomson's Construction (Example - (a|b) * a)

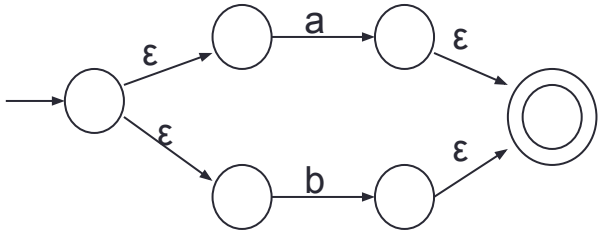
a:



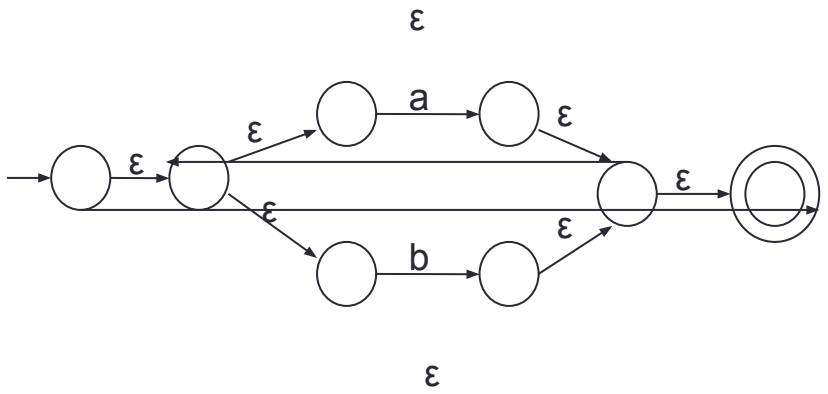
b:



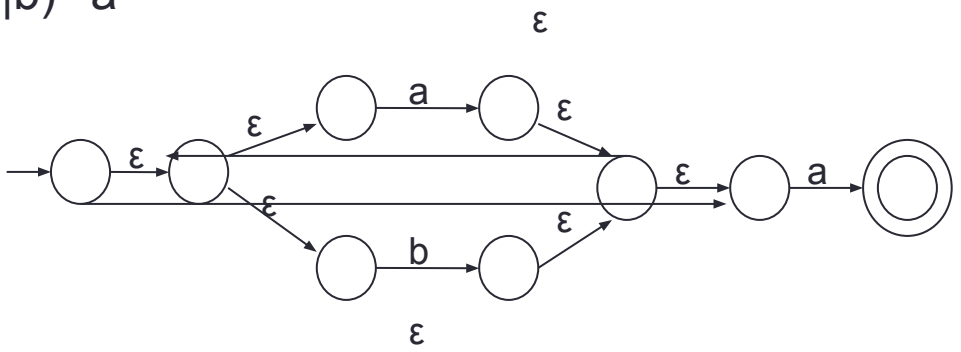
(a | b)



(a|b) *



(a|b) * a

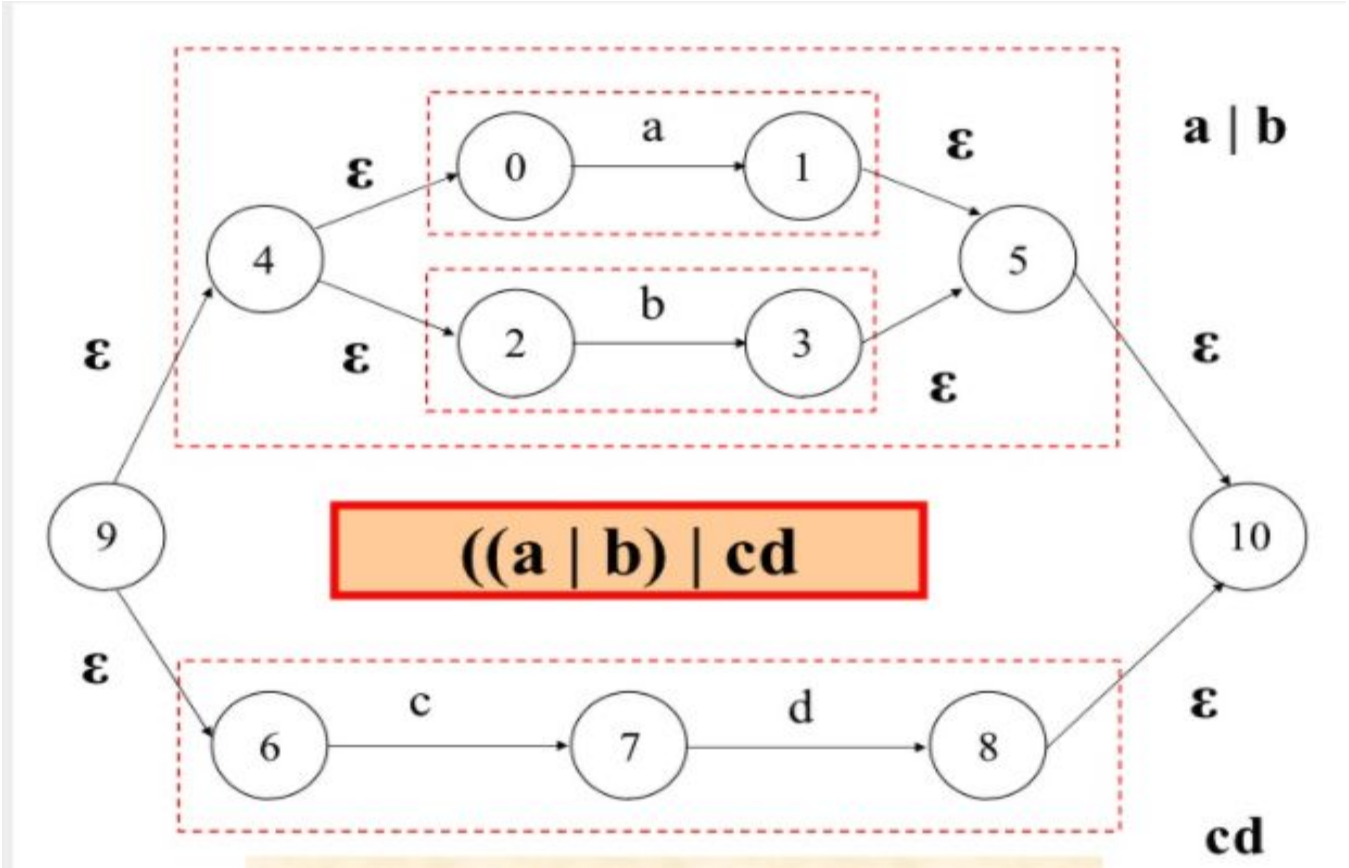


Thompson's Method

SRM INSTITUTE OF SCIENCE & TECHNOLOGY
(Deemed to be University u/s 3 of UGC Act, 1956)

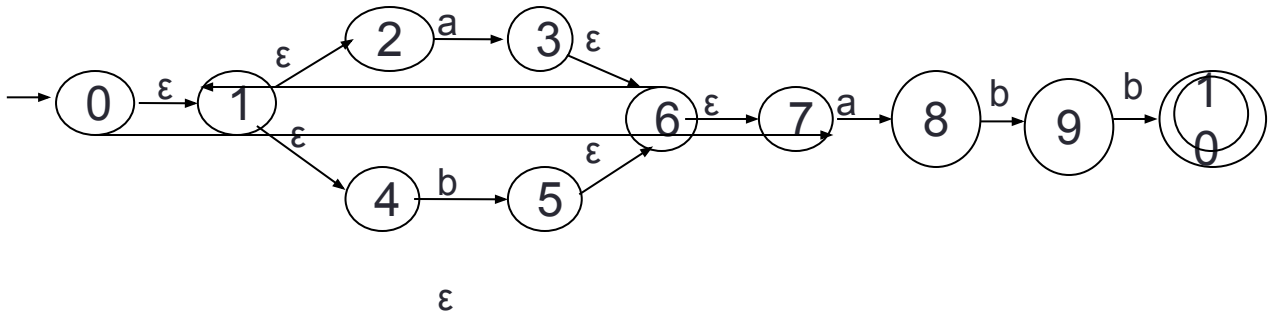
Construct an NFA that recognizes

((a | b) | cd)

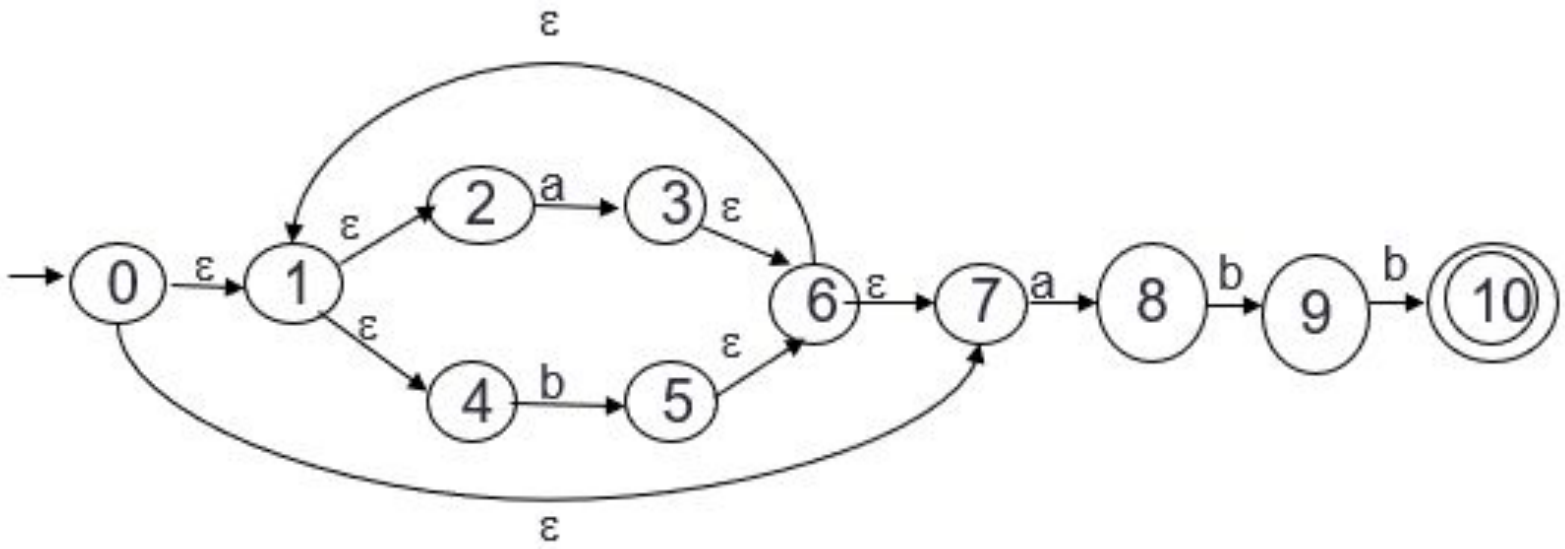


RE to NFA

$(a|b)^*abb$



$(a|b)^*abb$



RE to DFA –USING THOMPSONS SUBSET CONSTRUCTION

Given regular expression is converted into NFA..

Then. NFA is converted into DFA.

STEPS

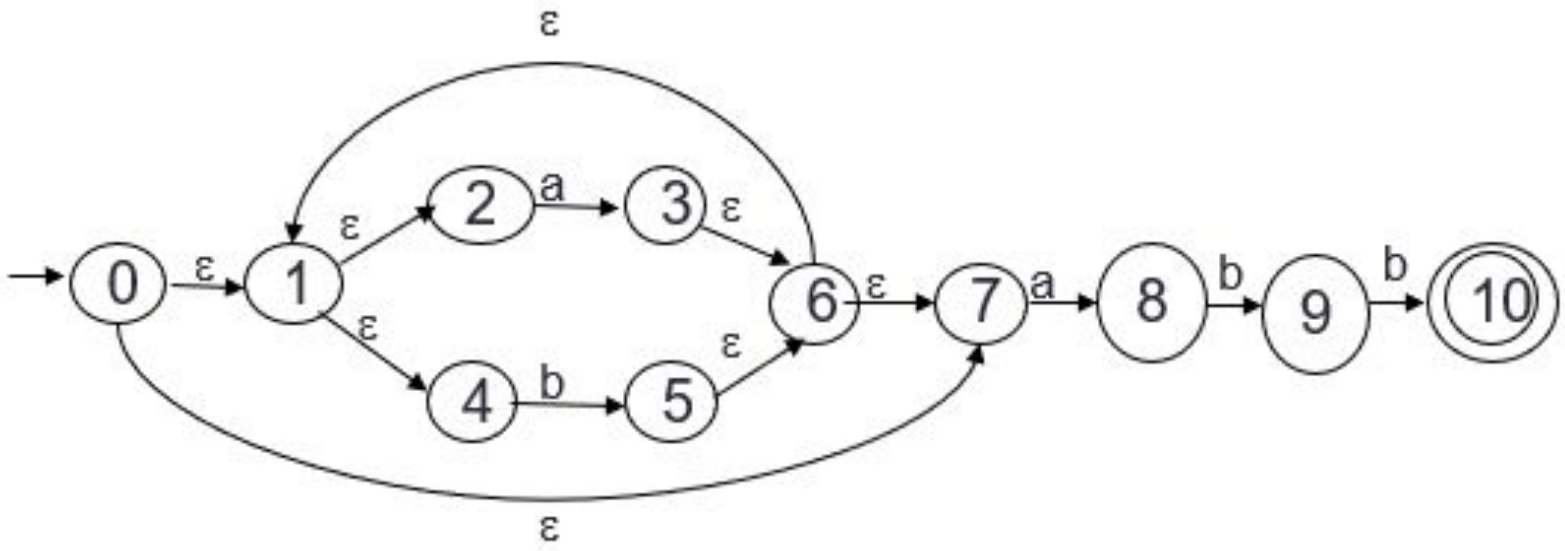
1. Convert into NFA using above rules for operators (union, concatenation and closure)and precedence.
2. Find Ɛ-closure of all states.
3. Start with epsilon closure of start state of NFA.
4. Apply the input symbols and find its epsilon closure.

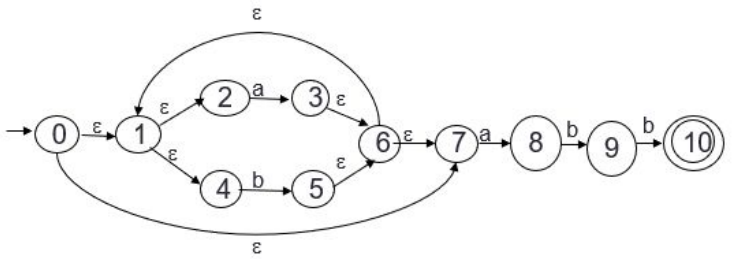
   Dtran[state, input symbol] = Ɛ-closure(move(state, input symbol))

   where Dtran transition function of DFA
5. Analyze the output state to find whether it is a new state.
6. If new state is found, repeat step 4 and step 5 until no more new states are found.
7. Construct the transition table for Dtran function.
8. Draw the transition diagram with start state as the s-closure (start state of NFA) andfinal state is the state that contains final state of NFA drawn.

STEP1: RE to NFA

(a|b)* abb





Step 2: Start with finding Ɛ-closure of state 0

Ɛ-closure(0) = {0,1,2,4,7} = **A**

Step 3: Apply input symbols a, b to A

Dtran(A, a) = Ɛ-closure(move(A, a))

= Ɛ-closure(move({0,1,2,4,7), a))= Ɛ-closure(3,8)= {3,6,7,1,2,4,8}= {1,2,3,4,6,7,8} = **B**

Dtran A, a=B

Dtran A, b] = Ɛ-closure(move(A,b))

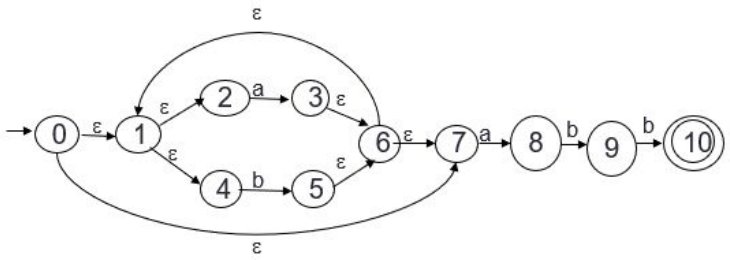
= Ɛ-closure(move({0,1,2,4,7), b))

= Ɛ-closure(5)

{5,6,7,1,2,4,7}

{1,2,4,5,6,7} = **C**

Dtran(A,b] =**C**



Step 4: Apply input symbols to new state B

Dtran B, a] = Ɛ-closure (move(B, a))

= Ɛ-closure(move({1,2,3,4,6,7,8}, a))

= Ɛ-closure(3,8)

= {1,2,3,4,6,7,8} = B

Dtran [B, a] =B

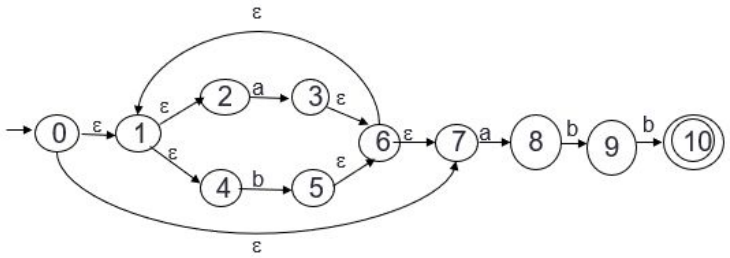
Dtran(B, b) = Ɛ-closure(move(B, b))

= Ɛ-closure(move({1,2,3,4,6,7,8},b))

= Ɛ-closure(5,9)

={1,2,4,5,6,7,9} =**D**

Dtran(B, 6] =D



Step 5: Apply input symbols to new state C

Dtran(C, a) = Ɛ-closure(move(C, a))

= Ɛ-closure(move({1,2,4,5,6,7), a))

= Ɛ-closure(3,8)

={1,2,3,4,6,7,8}= B

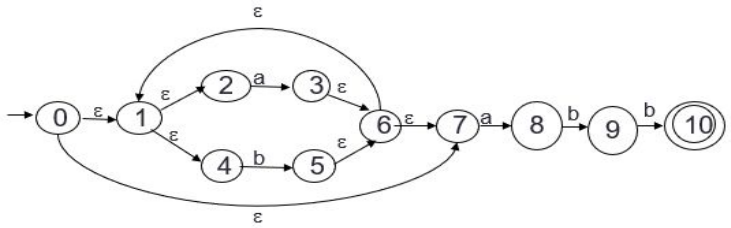
Dtran(C, a] =B

Dtran [C, b) = Ɛ-closure(move(C, b))

= Ɛ-closure(move({1,2,4,5,6,7),b))

= Ɛ-closure(5)= {1,2,4,5,6,7} = C

Dtran(C, b] =C



Step 6: Apply input symbols to new state D

Dtran[D, a] = Ɛ-closure(move(D, a))

= Ɛ-closure(move({1,2,4,5,6,7,9}, a))

= Ɛ-closure(3,8)

{1,2,3,4,6,7,8} = B

Dtran[D, a] = B

Dtran(D, b] = Ɛ-closure(move(D,b))

= Ɛ-closure(move({1,2,4,5,6,7,9},b))

= Ɛ-closure(5, 10)= {1,2,4,5,6,7,10} = **E**

Dtran[D, b] =E

Step 7: Apply input symbols to new state E

Dtran[E, a) =Ɛ -closure(move(E, a))

= Ɛ-closure(move({1,2,4,5,6,7,10), a))

= Ɛ-closure(3,8)

= {1,2,3,4,6,7,8} = B

Dtran [E, a] =B

Dtran [E, b] = E-closure(move(E, 6))

= E-closure(move({1,2,4,5,6,7,10),b))

= E-closure(5)= {1,2,4,5,6,7} = C

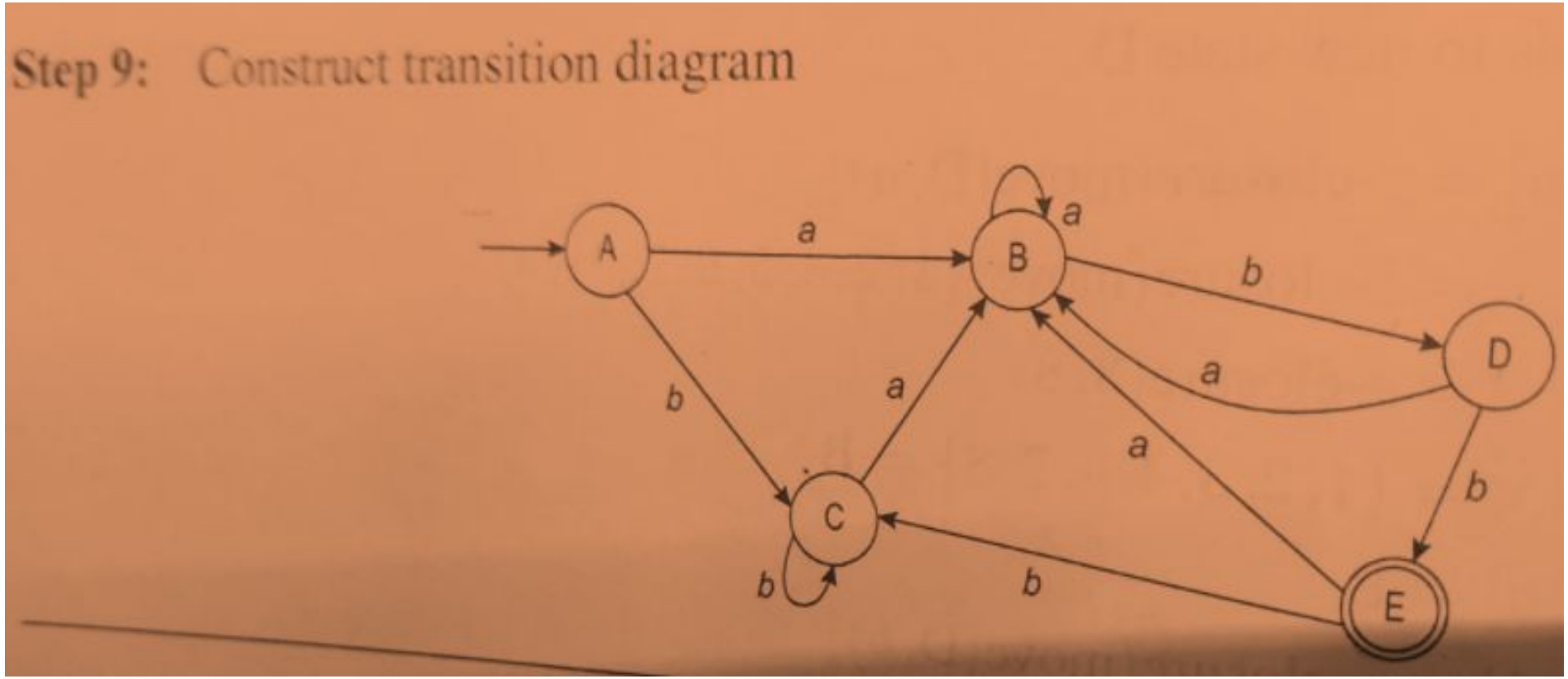
Dtran [E, b]=C

Step 8: Construct transition table

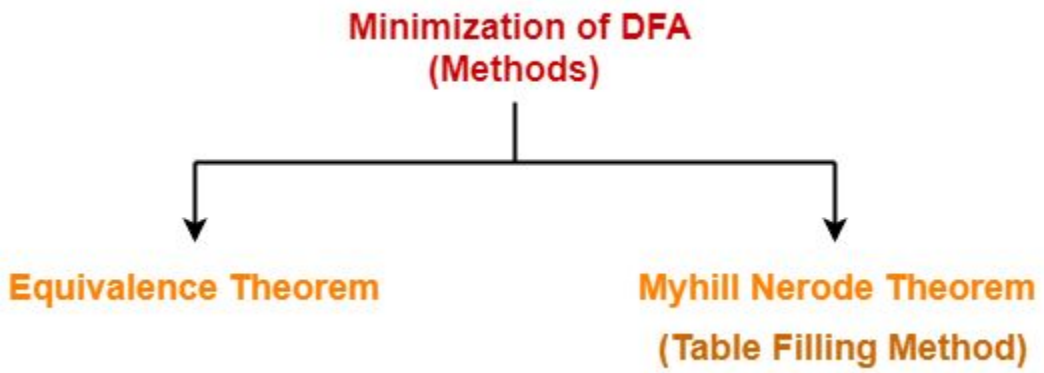
	a	*b*
→ A	B	C
B	B	D
C	B	C
D	B	E
*E	B	C

Note:

Step 9: Construct transition diagram



Minimization of DFA



Minimization of DFA

Step-01:

- Eliminate all the dead states and inaccessible states from the given DFA (if any).

Dead State

All those non-final states which transit to itself for all input symbols in ∑ are called as dead states.

Inaccessible State

All those states which can never be reached from the initial state are called as inaccessible states.

Minimization of DFA

Step-02:

- Draw a state transition table for the given DFA.
- Transition table shows the transition of all states on all input symbols in Σ.

Step-03:

Now, start applying equivalence theorem.

- Take a counter variable k and initialize it with value 0.
- Divide Q (set of states) into two sets such that one set contains all the non-final states and other set contains all the final states.
- This partition is called P_0.

Minimization of DFA

Step-04:

- Increment k by 1.
- Find P_k by partitioning the different sets of P_{k-1} .
- In each set of P_{k-1} , consider all the possible pair of states within each set and if the two states are distinguishable, partition the set into different sets in P_k.

> Two states q_1 and q_2 are distinguishable in partition P_k for any input symbol 'a',
>
> if $\delta(q_1, a)$ and $\delta(q_2, a)$ are in different sets in partition P_{k-1}.

Step-05:

- Repeat step-04 until no change in partition occurs.
- In other words, when you find $P_k = P_{k-1}$, stop.

Minimization of DFA

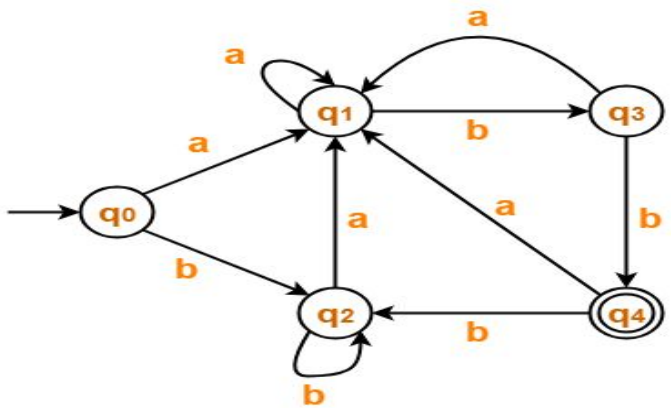
Step-06:

- All those states which belong to the same set are equivalent.
- The equivalent states are merged to form a single state in the minimal DFA.

Number of states in Minimal DFA

= Number of sets in P_k

Example-Minimization of DFA



Now using Equivalence Theorem, we have-

$P_0 = \{ q_0 , q_1 , q_2 , q_3 \} \{ q_4 \}$

$P_1 = \{ q_0 , q_1 , q_2 \} \{ q_3 \} \{ q_4 \}$

$P_2 = \{ q_0 , q_2 \} \{ q_1 \} \{ q_3 \} \{ q_4 \}$

$P_3 = \{ q_0 , q_2 \} \{ q_1 \} \{ q_3 \} \{ q_4 \}$

	a	b
→q0	q1	q2
q1	q1	q3
q2	q1	q2
q3	q1	*q4
*q4	q1	q2

Example-Minimization of DFA

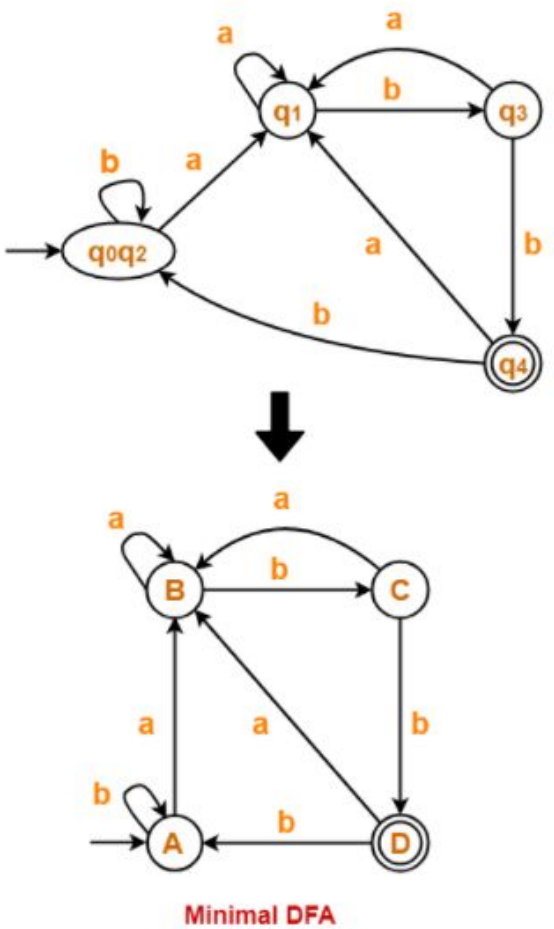
Now using Equivalence Theorem, we have-

$P_0 = \{ q_0 , q_1 , q_2 , q_3 \} \{ q_4 \}$

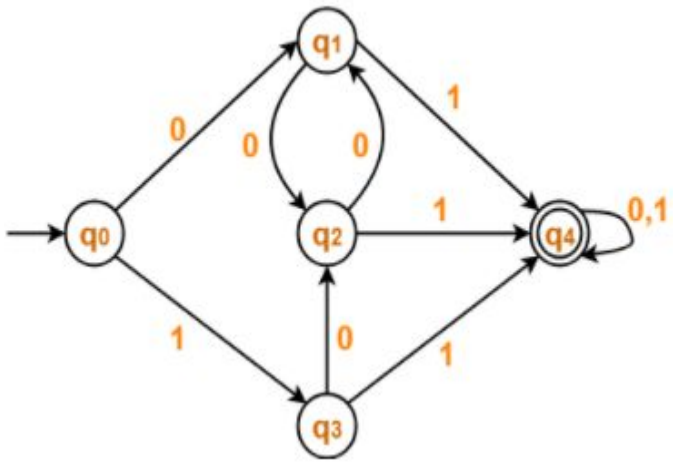
$P_1 = \{ q_0 , q_1 , q_2 \} \{ q_3 \} \{ q_4 \}$

$P_2 = \{ q_0 , q_2 \} \{ q_1 \} \{ q_3 \} \{ q_4 \}$

$P_3 = \{ q_0 , q_2 \} \{ q_1 \} \{ q_3 \} \{ q_4 \}$



Example-Minimization of DFA



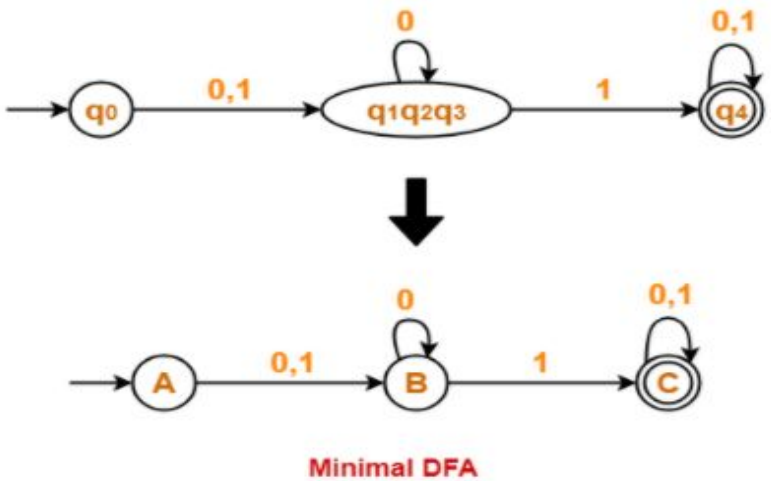
	0	1
→q0	q1	q3
q1	q2	*q4
q2	q1	*q4
q3	q2	*q4
*q4	*q4	*q4

Now using Equivalence Theorem, we have-

$P_0 = \{ q_0 , q_1 , q_2 , q_3 \} \{ q_4 \}$

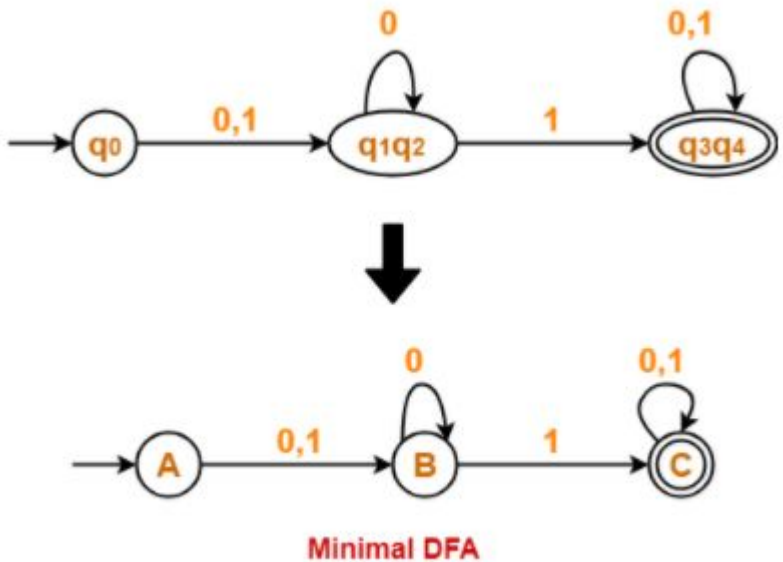
$P_1 = \{ q_0 \} \{ q_1 , q_2 , q_3 \} \{ q_4 \}$

$P_2 = \{ q_0 \} \{ q_1 , q_2 , q_3 \} \{ q_4 \}$



Example-Minimization of DFA

	0	1
→q0	q1	q2
q1	q2	*q3
q2	q2	*q4
*q3	*q3	*q3
*q4	*q4	*q4



Now using Equivalence Theorem, we have-

$P_0 = \{ q_0 , q_1 , q_2 \} \{ q_3 , q_4 \}$

$P_1 = \{ q_0 \} \{ q_1 , q_2 \} \{ q_3 , q_4 \}$

$P_2 = \{ q_0 \} \{ q_1 , q_2 \} \{ q_3 , q_4 \}$

Regular Expression to DFA (Direct Method)

- We may convert a regular expression into a DFA (without creating a NFA first).

1. First we augment the given regular expression by concatenating it with a special symbol #.

   r → (r)# augmented regular expression

2. Then, construct a syntax tree from the augmented regular expression (r)#

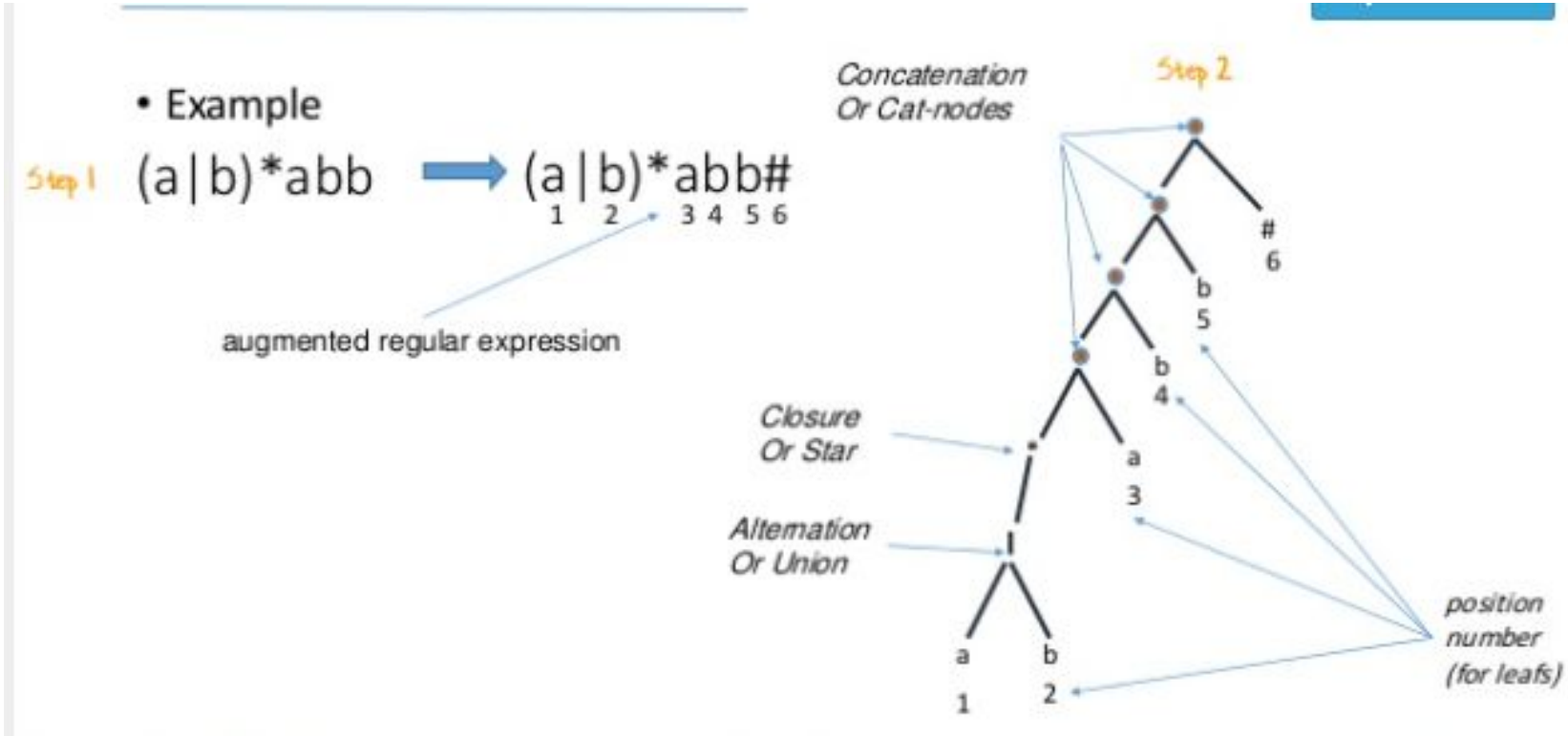
3. Leaves in a syntax tree are labeled by an alphabet symbols (plus #) or by the empty string, and inner nodes will be the operators in the augmented regular expression.

4. Then each alphabet symbol (plus #) will be numbered (position numbers).

5. Finally, compute four functions: *nullable*, *firstpos*, *lastpos* and *followpos*.

Regular Expression to DFA (Direct Method)- Example

- Regular Expression: (a/b)*abb
- Augmented Grammar : (a/b)*abb# =⍰ (a/b)*.a.b.b.#



Regular Expression to DFA (Direct Method)- Example

- There are four functions have to be computed from syntax tree

1. *Nullable(n):* is true for a syntax tree node n if the subexpression represented by n has ε in its languages.
2. *Firstpos(n):* is the set of the positions in the subtree that correspond to the first symbols of strings generated by the sub-expression rooted by n.
3. *Lastpos(n):* is the set of the positions in the subtree that correspond to the last symbols of strings generated by the sub-expression rooted by n.
4. *Followpos(i):* is the set of positions that can follow the position i in the tree in the strings generated by the augmented regular expression.

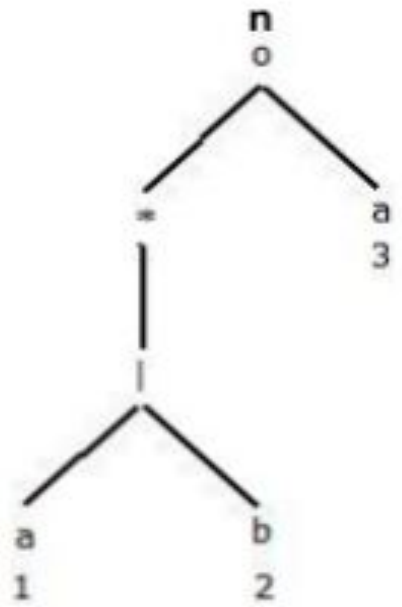
Computation of Nullable, Firstpos, LastPos:

Node n	*nullable*(n)	*firstpos*(n)	*lastpos*(n)
Leaf ε	true	$\varnothing$	$\varnothing$
Leaf i	false	$\{i\}$	$\{i\}$
	*nullable*(c_1) or *nullable*(c_2)	*firstpos*(c_1) $\cup$ *firstpos*(c_2)	*lastpos*(c_1) $\cup$ *lastpos*(c_2)
	*nullable*(c_1) and *nullable*(c_2)	**if** *nullable*(c_1) **then** *firstpos*(c_1) $\cup$ *firstpos*(c_2) **else** *firstpos*(c_1)	**if** *nullable*(c_2) **then** *lastpos*(c_1) $\cup$ *lastpos*(c_2) **else** *lastpos*(c_2)
	true	*firstpos*(c_1)	*lastpos*(c_1)

Node n	nullable *(n)*	firstpos *(n)*	lastpos *(n)*
A leaf labeled Ɛ	True	Ø	Ø
A leaf with position i	False	{i}	{i}
An *or* node *n* = $c_1 \mid c_2$	Nullable (c_1) or Nullable (c_2)	firstpos (c_1) U firstpos (c_2)	lastpos (c_1) U lastpos (c_2)
A cat node *n* = c_1c_2	Nullable (c_1) and Nullable (c_2)	If (Nullable (c_1)) firstpos (c_1) U firstpos (c_2) else firstpos (c_1)	If (Nullable (c_2)) lastpos (c_1) U lastpos (c_2) else lastpos (c_1)
A star node *n* = c_1^*	True	firstpos (c_1)	lastpos (c_1)

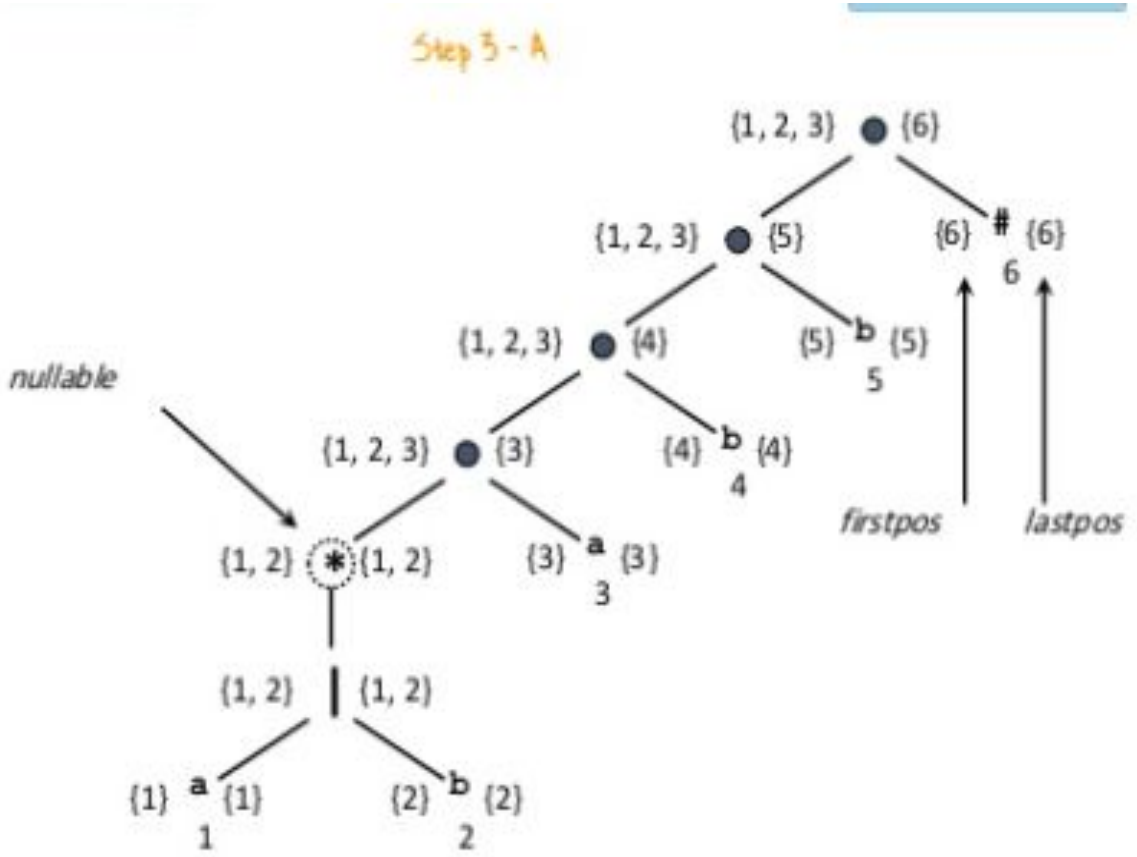
Example:

- (a|b)* a
  1 2 3
- nullable(n)=false
- firstpos(n)={1,2,3}
- lastpos(n)={3}
- followpos(1)={1,2,3}



Direct Method

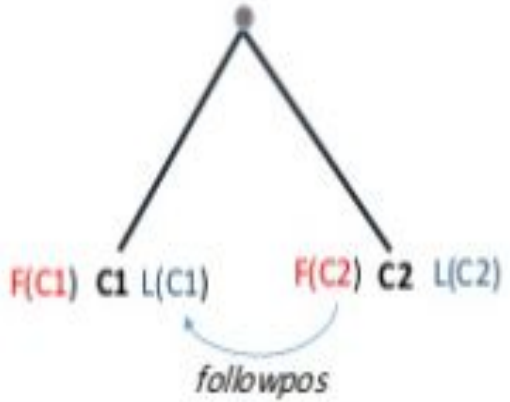
(a|b)*abb#
1 2 3 4 5 6



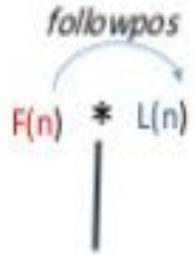
Direct Method

Followpos can be computed as following

- (rule 1) if n is a cat-node c1 c2
  for every position i in lastpos(c1), then
  all positions in firstpos(c2) are in followpos(i)



- (rule 2) if n is a star-node
  if i is a position in lastpos(n), then
  all positions in firstpos(n) are in followpos(i)



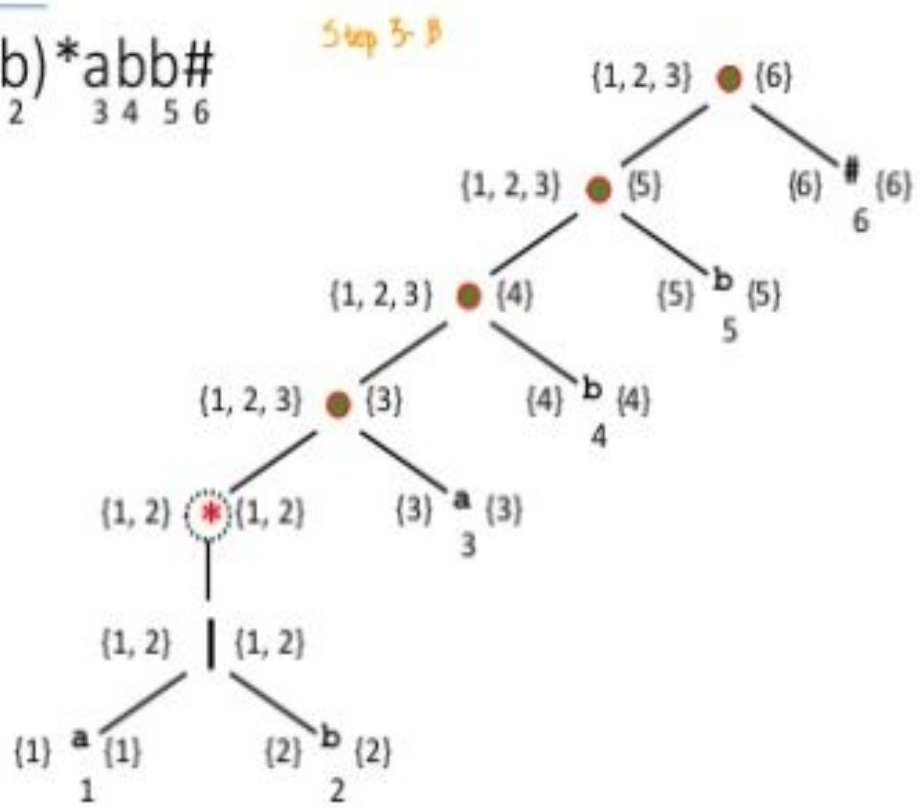
Direct Method

for each node n in the tree **do**
    if n is a cat-node with left child c_1 and right child c_2 **then**
        for each i in $lastpos(c_1)$ **do**
            $followpos(i) := followpos(i) \cup firstpos(c_2)$
        end do
    else if n is a star-node
        for each i in $lastpos(n)$ **do**
            $followpos(i) := followpos(i) \cup firstpos(n)$
        end do
    end if
end do

Direct Method

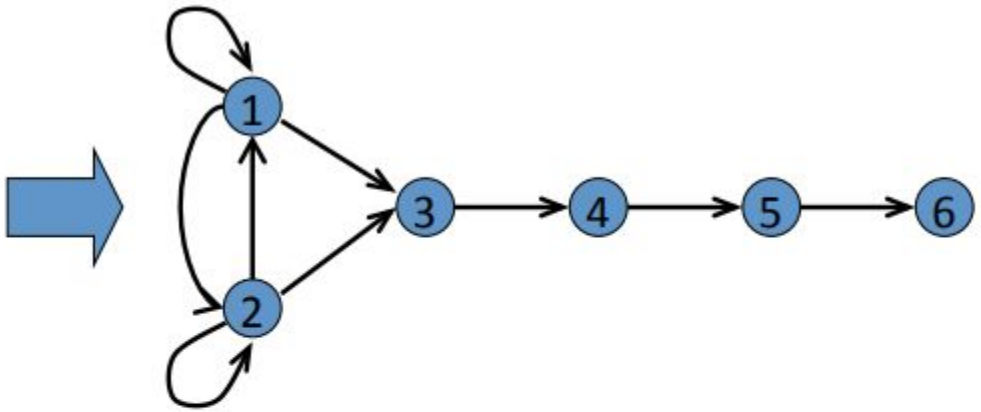
$$(a|b)^*abb\#$$
$$1\ \ 2\ \ \ 3\ 4\ 5\ 6$$

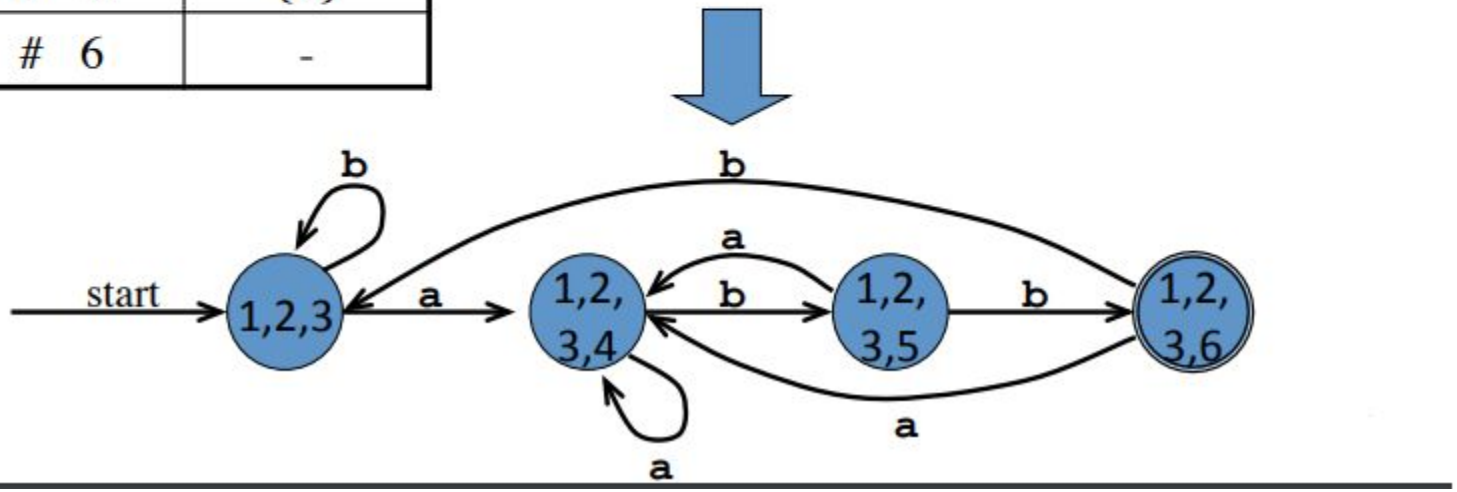
- Applying rule 1
- followpos(1) incl.{3}
- followpos(2) incl.{3}
- followpos(3) incl.{4}
- followpos(4) incl.{5}
- followpos(5) incl.{6}
- Applying rule 2
- followpos(1) incl.{1,2}
- followpos(2) incl.{1,2}



Direct Method

Node	*followpos*
a 1	{1, 2, 3}
b 2	{1, 2, 3}
a 3	{4}
b 4	{5}
b 5	{6}
# 6	-



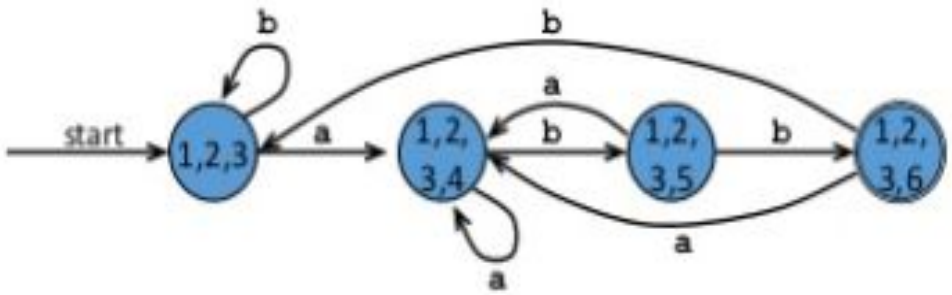


Direct Method

Step 4

Node	*followpos*
1	{1, 2, 3}
2	{1, 2, 3}
3	{4}
4	{5}
5	{6}
6	-

A=firstpos(n0)={1,2,3}
Move[A,a]=
followpos(1) U followpos(3)= {1,2,3,4}=B
o Move[A,b]=
followpos(2)={1,2,3}=A
o Move[B,a]=
followpos(1) U followpos(3)=B
o Move[B,b]=
followpos(2) U followpos(4)={1,2,3,5}=C



(a|b)*abb#
1 2 3 4 5 6

Node n	nullable *(n)*	firstpos *(n)*	lastpos *(n)*
A leaf labeled Ɛ	True	Ø	Ø
A leaf with position i	False	{i}	{i}
An *or* node *n* = $c_1 \mid c_2$	Nullable (c_1) or Nullable (c_2)	firstpos (c_1) U firstpos (c_2)	lastpos (c_1) U lastpos (c_2)
A cat node *n* = c_1c_2	Nullable (c_1) and Nullable (c_2)	If (Nullable (c_1)) firstpos (c_1) U firstpos (c_2) else firstpos (c_1)	If (Nullable (c_2)) lastpos (c_1) U lastpos (c_2) else lastpos (c_1)
A star node *n* = c_1^*	True	firstpos (c_1)	lastpos (c_1)

Optimisation of DFA from Regular Expression [Direct method] $((\epsilon|a)b^*)^*\#$

STEP 1 :

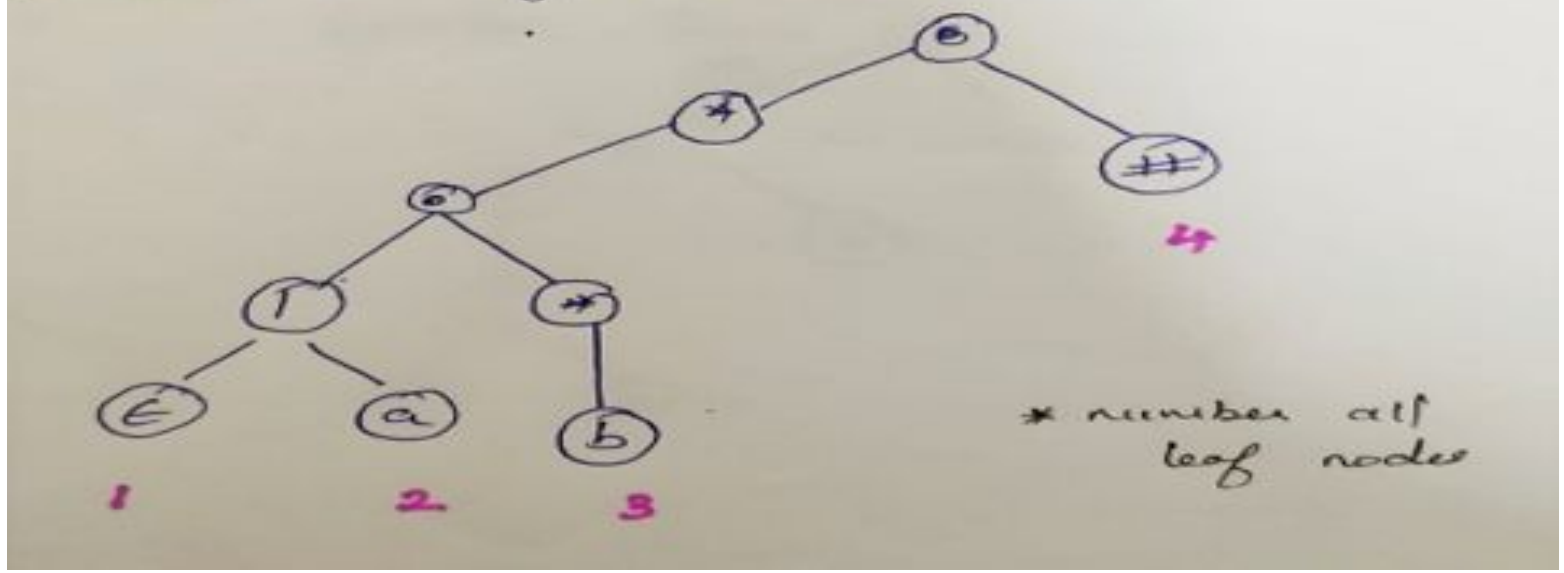
Augumented Regular Expression

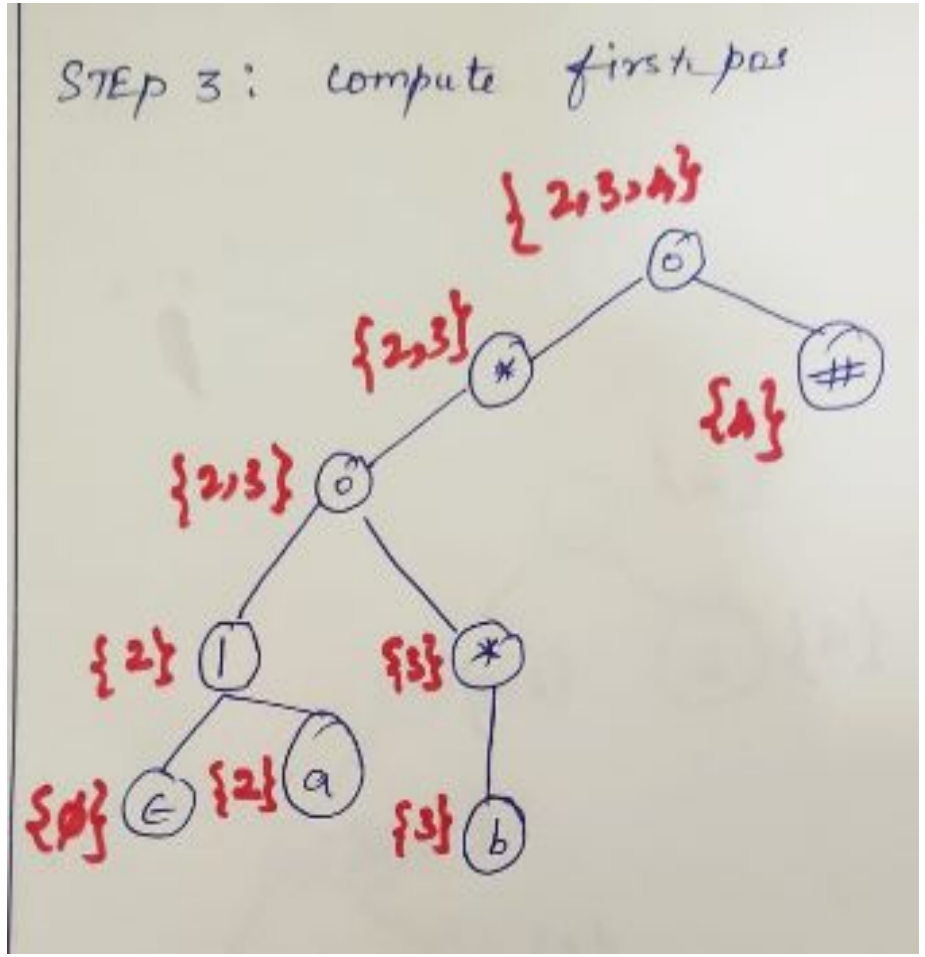
$(r).\#$

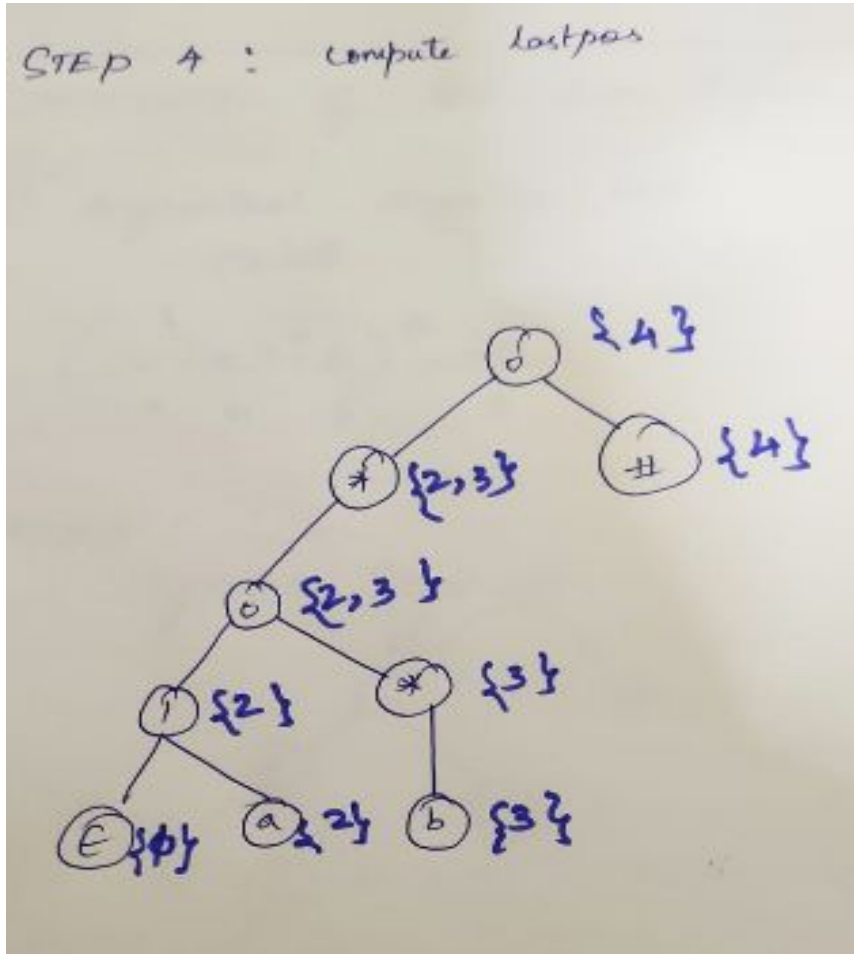
$((\epsilon|a).b^*)^*.\#$

1 2 3 4

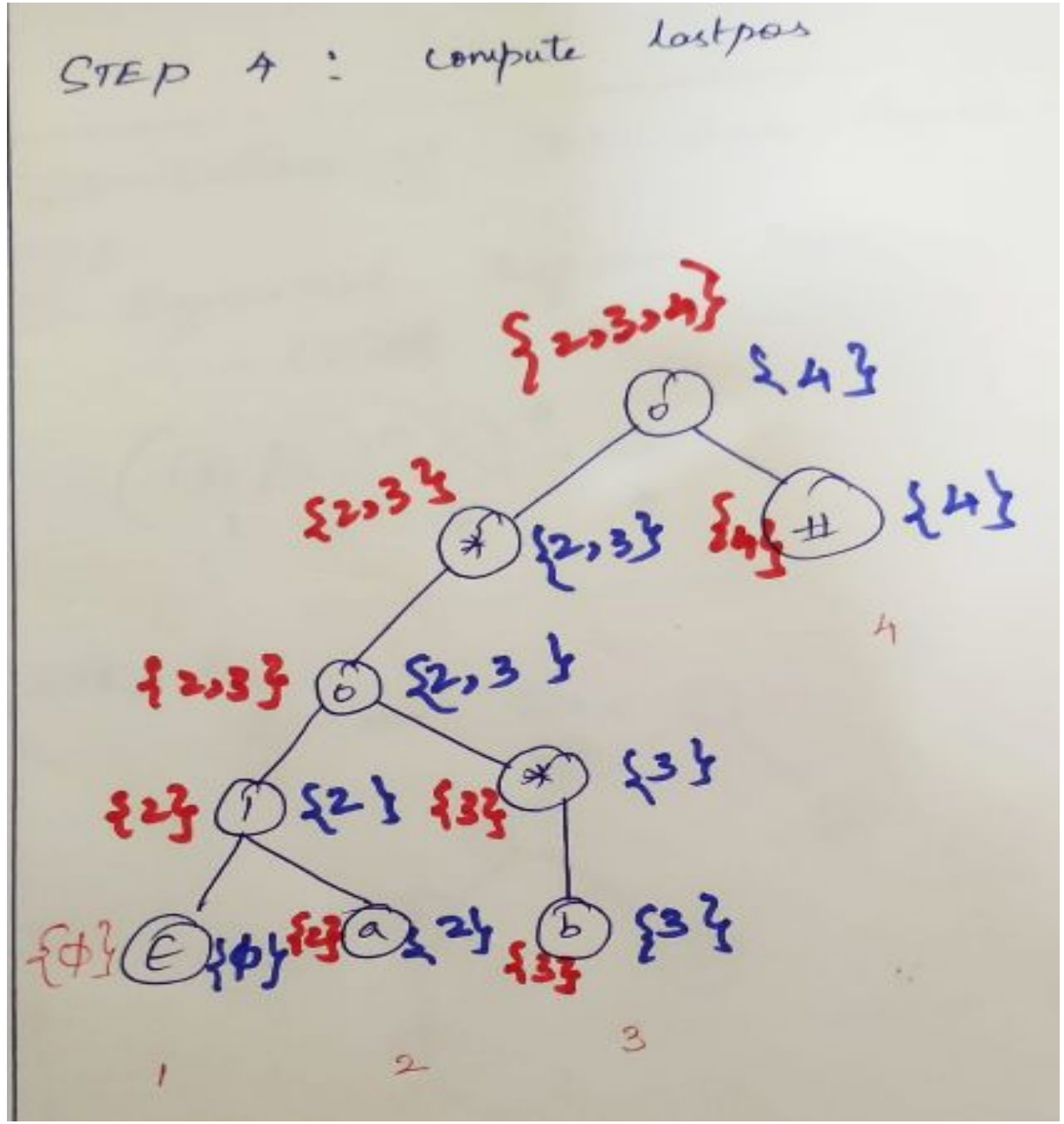
STEP 2 : syntax tree




STEP 3 : compute firstpos
{2,3,4}
{2,3}
{4}
{2,3}
{2}
{3}
{ϕ}
{2}
{3}


STEP 4 : compute lastpos
{4}
{4}
{2,3}
{2,3}
{2}
{3}
{ϕ}
{2}
{3}

STEP 4 : compute lastpos



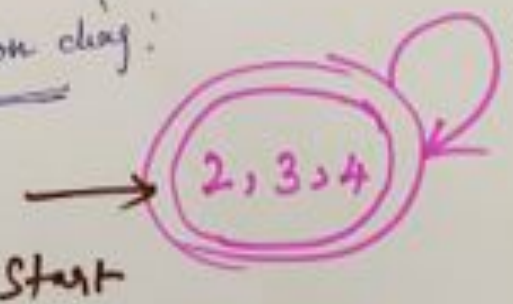
STEP 5 :

$((\epsilon | a) b^*)^* . \#$
1 2 3 4

Computing followpos

	position i	followpos(i)
ϵ	1	ϕ
a	2	{2, 3, 4}
b	3	{2, 3, 4}
#	4	ϕ

Transition diag :



Transition table for input :

State	a	b
A	A	A

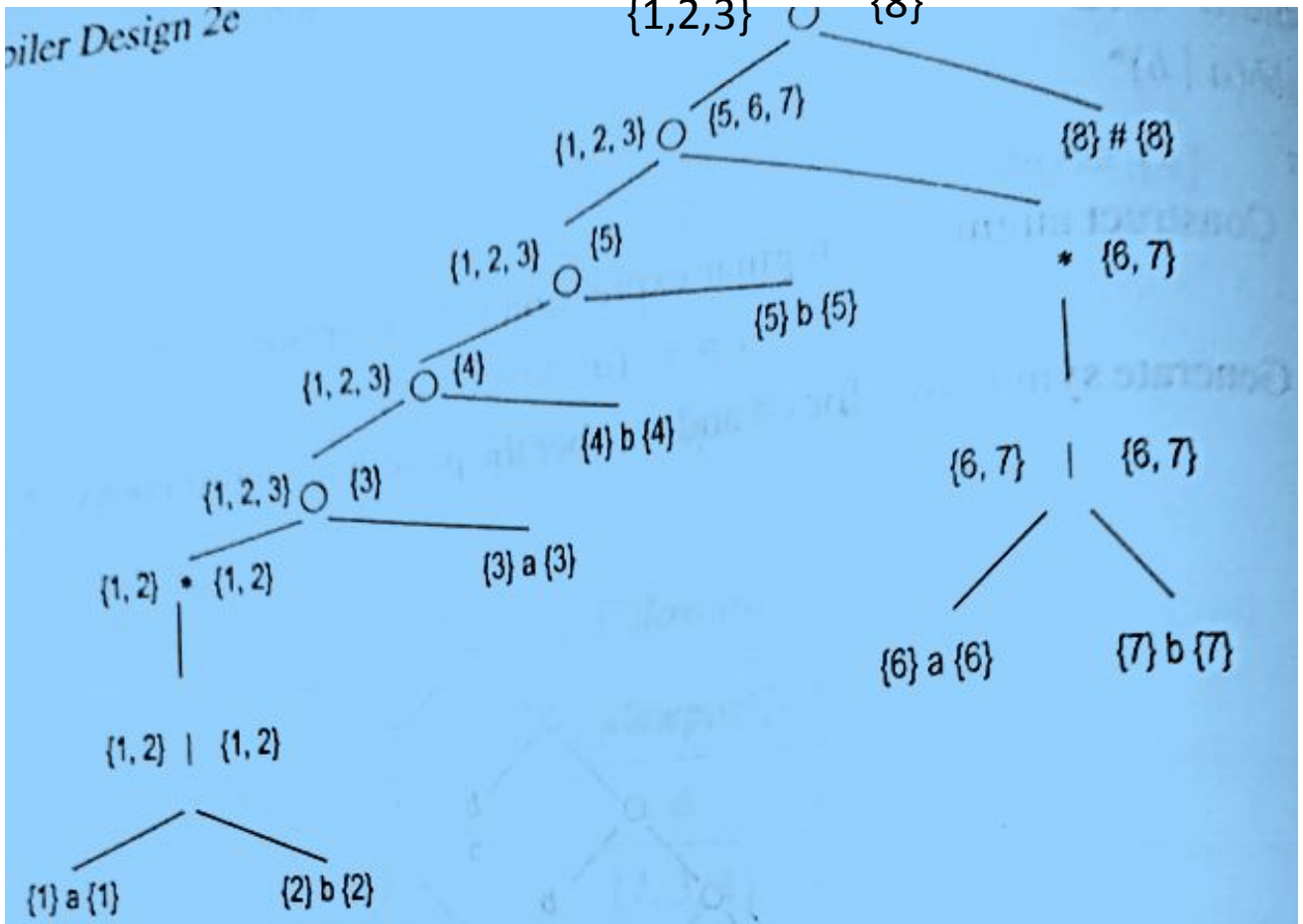
Direct method RE to DFA

(a|b)*abb(a|b)*

Find firstpos,lastpos,followpos

Draw DFA

{1,2,3} {8}

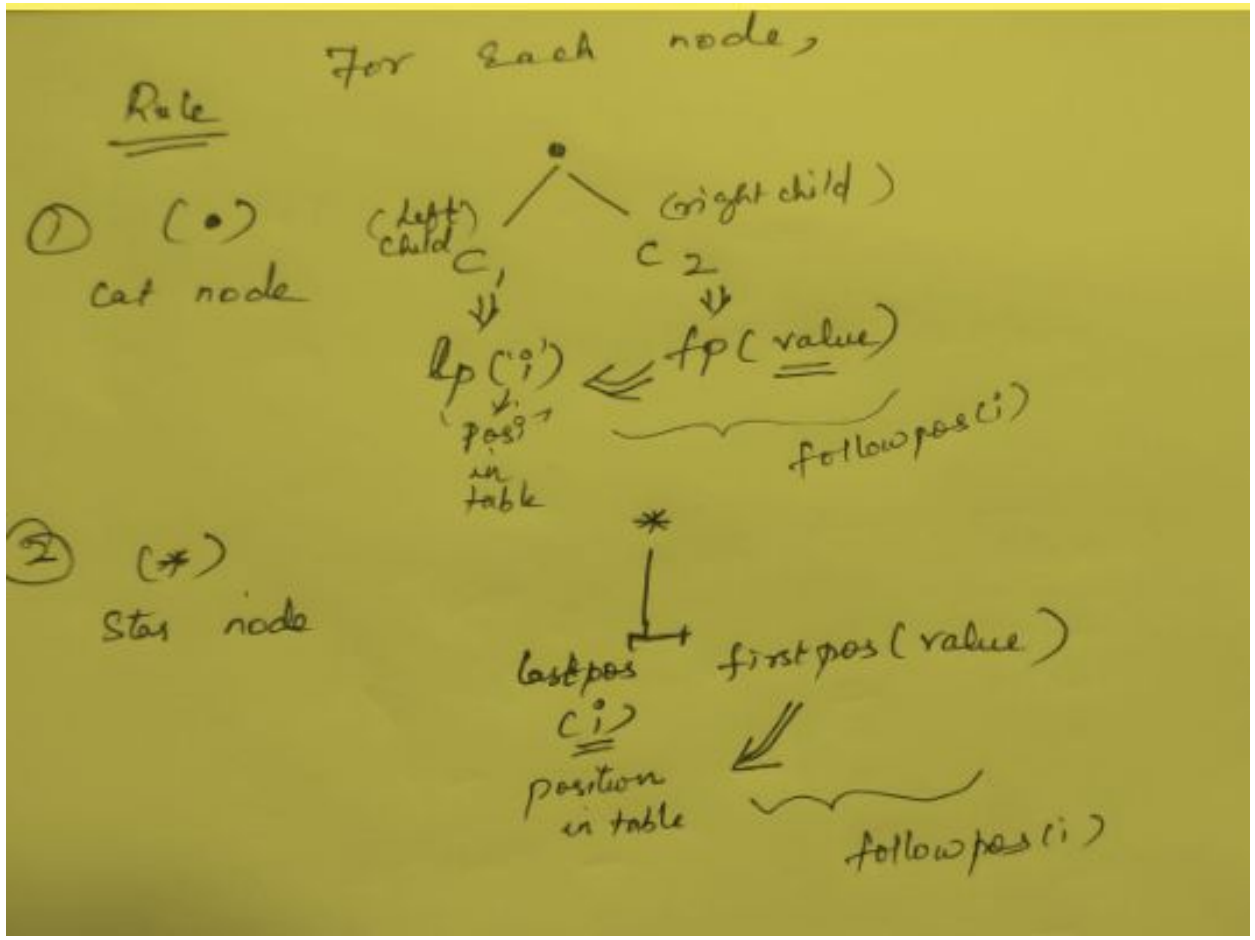


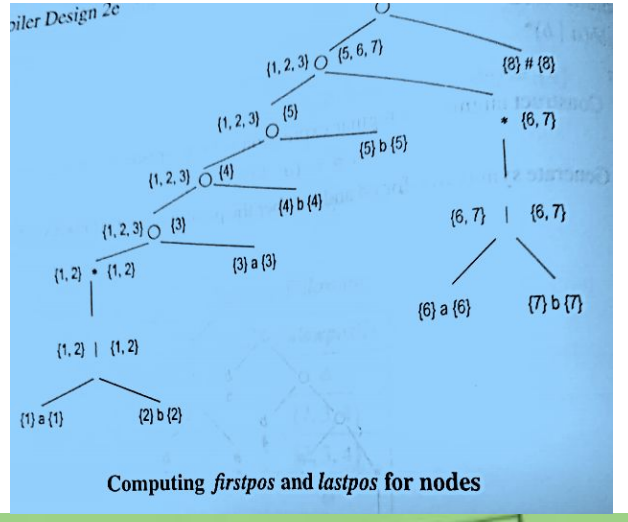
Computing *firstpos* and *lastpos* for nodes

Computing *followpos*

Position *i*	*followpos(i)*
a 1	{1, 2, 3}
b 2	{1, 2, 3}
a 3	{4}
b 4	{5}
b 5	{6, 7, 8}
a 6	{6, 7, 8}
b 7	{6, 7, 8}
# 8	ϕ

Finding followpos by rules





Computing *firstpos* and *lastpos* for nodes

position (i)		followpos(i)
a	1	{1, 2, 3}
b	2	{1, 2, 3}
a	3	{4}
b	4	{5}
b	5	{6, 7, 8}
a	6	{6, 7, 8}
b	7	{6, 7, 8}
#	8	ϕ

Step 6: Constructing Dtran function

Start state is the $firstpos$(root node of T) $= \{1,2,3\}$

Let A $= \{1,2,3\}$

Dtran(A, a) $= followpos(1) \cup followpos(3)$ [since a is found in positions 1 and 3]
$= \{1,2,3\} \cup \{4\}$
$= \{1,2,3,4\} \Rightarrow$ B

Dtran(A, a) = B

Dtran(A, b) $= followpos(2)$ [since b is found in position 2]
$= \{1,2,3\} \Rightarrow$ A

Dtran(A, b) = A

Dtran(B, a) $= followpos(1) \cup followpos(3)$ [since a is found in positions 1 and 3]
$= \{1,2,3\} \cup \{4\}$
$= \{1,2,3,4\} \Rightarrow$ B

Dtran(B, a) = B

Dtran(B, b) $= followpos(2) \cup followpos(4)$ [since b is found in positions 2 and 4]
$= \{1,2,3\} \cup \{5\}$
$= \{1,2,3,5\} \Rightarrow$ C

Dtran(B, b) = C

Dtran(C, a) $= followpos(1) \cup followpos(3)$ [since a is found in positions 1 and 3]
$= \{1,2,3\} \cup \{4\}$
$= \{1,2,3,4\} \Rightarrow$ B

Dtran(C, a) = B

Dtran(C, b) $= followpos(2) \cup followpos(5)$ [since b is found in positions 2 and 5]
$= \{1,2,3\} \cup \{6,7,8\}$
$= \{1,2,3,6,7,8\} \Rightarrow$ D

Dtran(C, b) = D

Dtran(D, a) $= followpos(1) \cup followpos(3) \cup followpos(6)$
[since a is found in positions 1, 3 and 6]
$= \{1,2,3\} \cup \{4\} \cup \{6,7,8\}$
$= \{1,2,3,4,6,7,8\} \Rightarrow$ E

Dtran(D, a) = E

Dtran(D, b) $= followpos(2) \cup followpos(7)$ [since b is found in positions 2 and 7]
$= \{1,2,3\} \cup \{6,7,8\}$
$= \{1,2,3,6,7,8\} \Rightarrow$ D

Dtran(D, b) = D

Dtran(E, a) $= followpos(1) \cup followpos(3) \cup followpos(6)$
[since a is found in positions 1, 3 and 6]
$= \{1,2,3\} \cup \{4\} \cup \{6,7,8\}$
$= \{1,2,3,4,6,7,8\} \Rightarrow$ E

$\text{Dtran}(E, a) = E$

$\text{Dtran}(E, b) = followpos(2) \cup followpos(4) \cup followpos(7)$

[since b is found in positions 2, 4 and 7]

$= \{1, 2, 3\} \cup \{5\} \cup \{6, 7, 8\}$

$= \{1, 2, 3, 5, 6, 7, 8\} \Rightarrow F$

$\text{Dtran}(E, b) = F$

$\text{Dtran}(F, a) = followpos(1) \cup followpos(3) \cup followpos(6)$

[since a is found in positions 1, 3 and 6]

$= \{1, 2, 3\} \cup \{4\} \cup \{6, 7, 8\}$

$= \{1, 2, 3, 4, 6, 7, 8\} \Rightarrow E$

$\text{Dtran}(F, a) = E$

$\text{Dtran}(F, b) = followpos(2) \cup followpos(5) \cup followpos(7)$

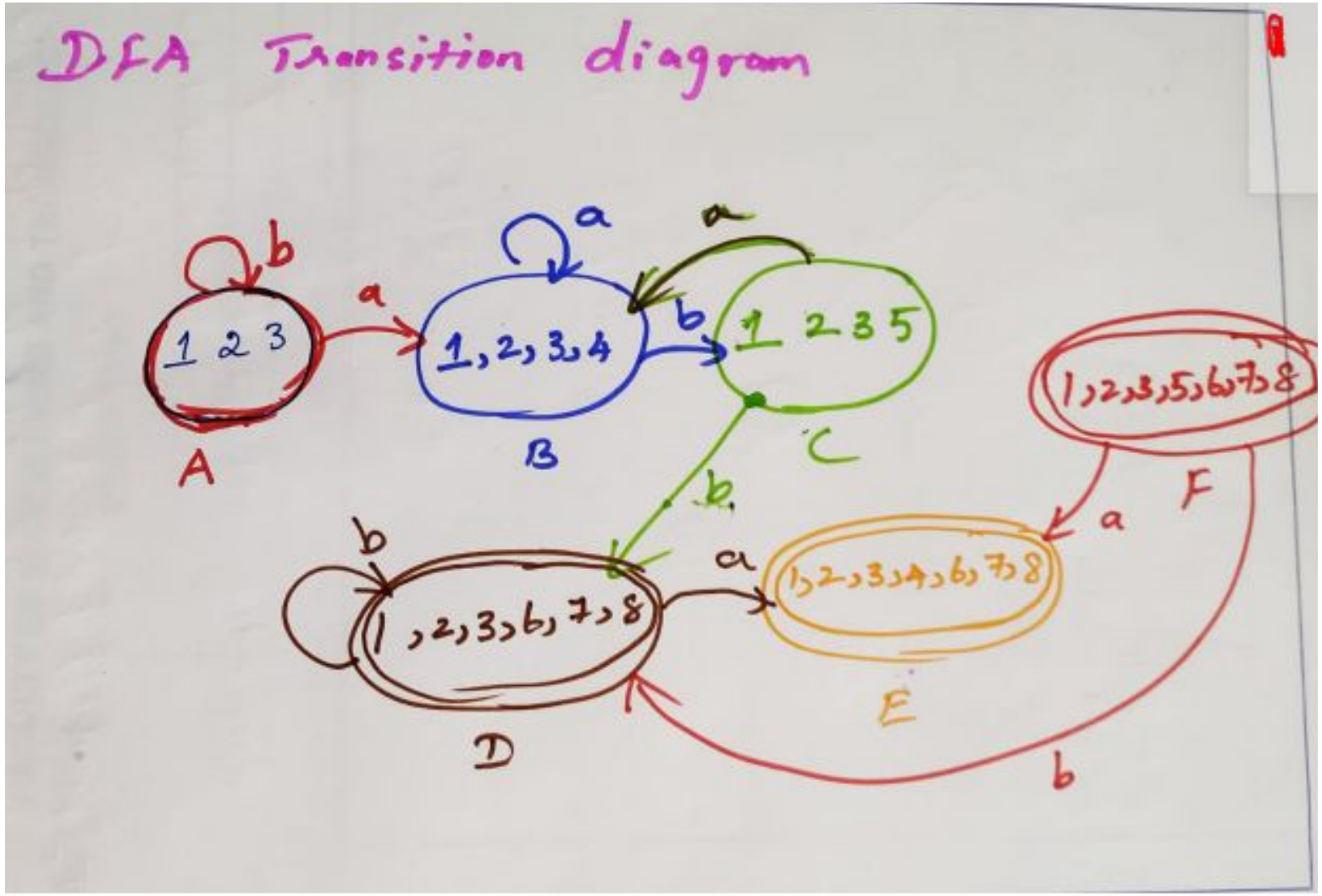
[since b is found in positions 2, 5 and 7]

$= \{1, 2, 3\} \cup \{6, 7, 8\} \cup \{6, 7, 8\}$

$= \{1, 2, 3, 6, 7, 8\} \Rightarrow D$

$\text{Dtran}(F, b) = D$

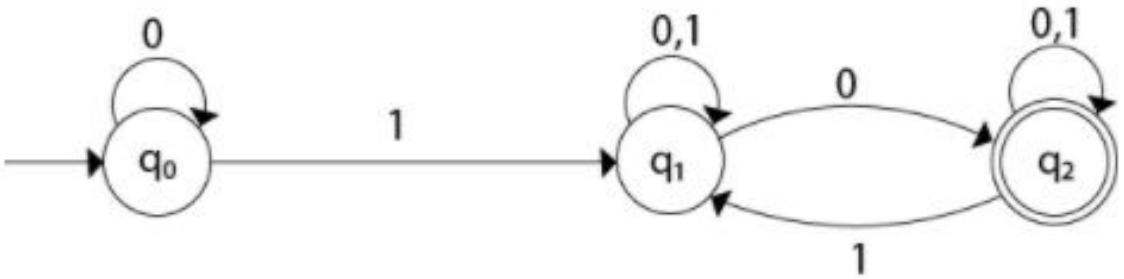
States \ Input	a	b
→A	B	A
B	B	C
C	B	D
*D	E	D
*E	E	F
*F	E	D


DFA Transition diagram
1 2 3
A
1,2,3,4
B
1 2 3 5
C
1,2,3,5,6,7,8
F
1,2,3,6,7,8
D
1,2,3,4,6,7,8
E
a
b

NFA TO DFA (SUBSET CONSTRUCTION)

Example 1:

Convert the given NFA to DFA.



Solution: For the given transition diagram we will first construct the transition table.

State	0	1
→q0	q0	q1
q1	{q1, q2}	q1
*q2	q2	{q1, q2}

Now we will obtain δ' transition for state q0.

$$\delta'([q0], 0) = [q0]$$
$$\delta'([q0], 1) = [q1]$$

The δ' transition for state q1 is obtained as:

$$\delta'([q1], 0) = [q1, q2] \quad \text{(new state generated)}$$
$$\delta'([q1], 1) = [q1]$$

The δ' transition for state q2 is obtained as:

$$\delta'([q2], 0) = [q2]$$
$$\delta'([q2], 1) = [q1, q2]$$

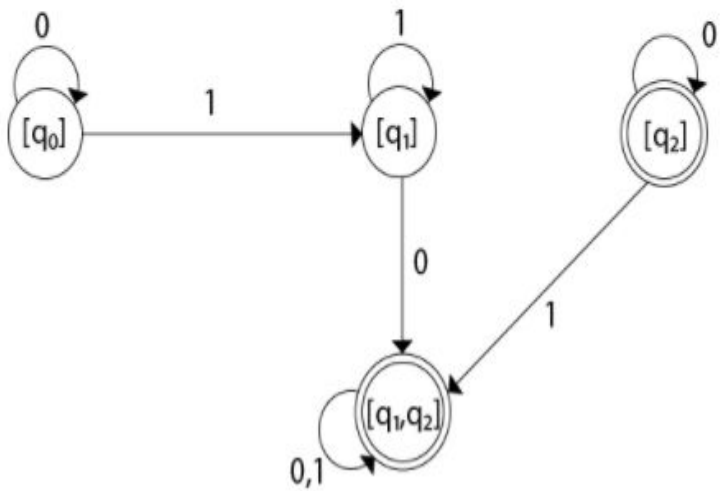
Now we will obtain δ' transition on [q1, q2].

$$\begin{aligned}\delta'([q1, q2], 0) &= \delta(q1, 0) \cup \delta(q2, 0)\\ &= \{q1, q2\} \cup \{q2\}\\ &= [q1, q2]\\ \delta'([q1, q2], 1) &= \delta(q1, 1) \cup \delta(q2, 1)\\ &= \{q1\} \cup \{q1, q2\}\\ &= \{q1, q2\}\\ &= [q1, q2]\end{aligned}$$

The state [q1, q2] is the final state as well because it contains a final state q2. The transition table for the constructed DFA will be:

State	0	1
→[q0]	[q0]	[q1]
[q1]	[q1, q2]	[q1]
*[q2]	[q2]	[q1, q2]
*[q1, q2]	[q1, q2]	[q1, q2]

The Transition diagram will be:

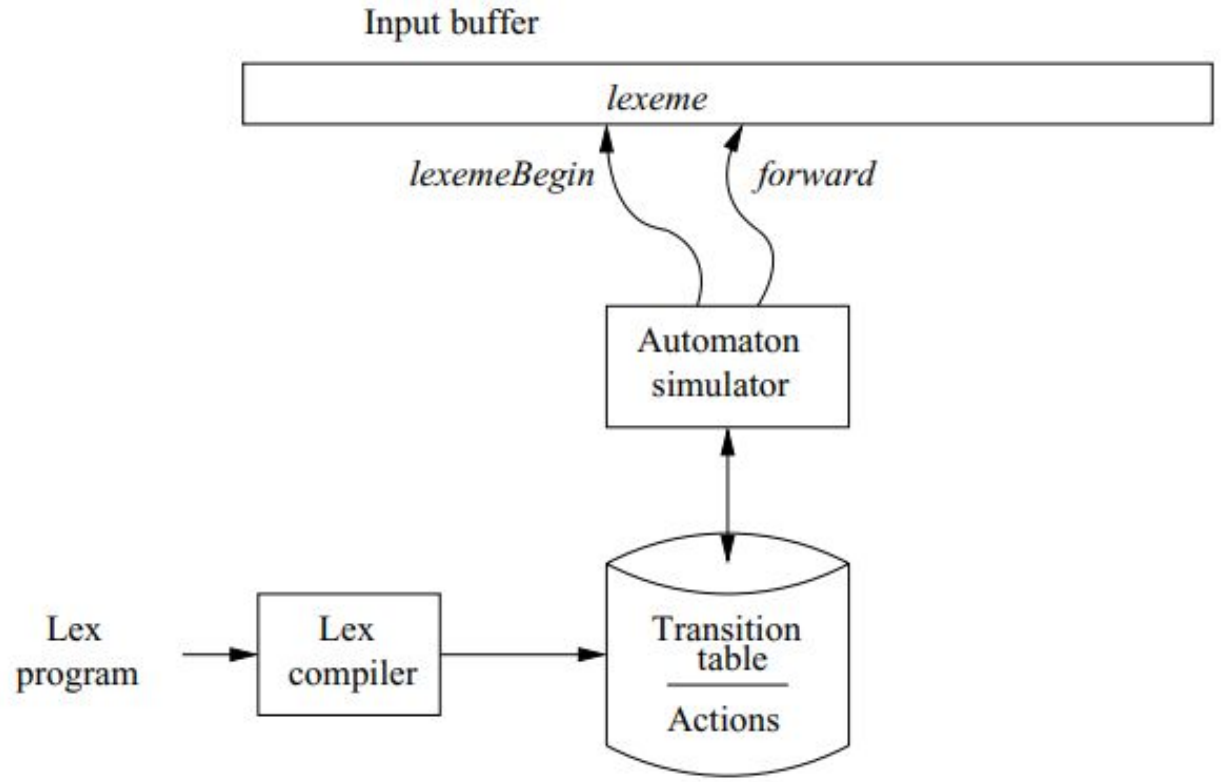


The state q2 can be eliminated because q2 is an unreachable state.

Design of Lexical Analyzer

- Lexical analyzer can either be generated by NFA or by DFA.
- DFA is preferable in the implementation of lex.

STRUCTURE OF LEXICAL ANALYZER



A Lex program is turned into a transition table and actions, which are used by a finite automaton simulator

These components are:

1. A transition table for the automaton.
2. Those functions that are passed directly through Lex to the output.
3. The actions from the input program, which appear as fragments of code to be invoked at the appropriate time by the automaton simulator.

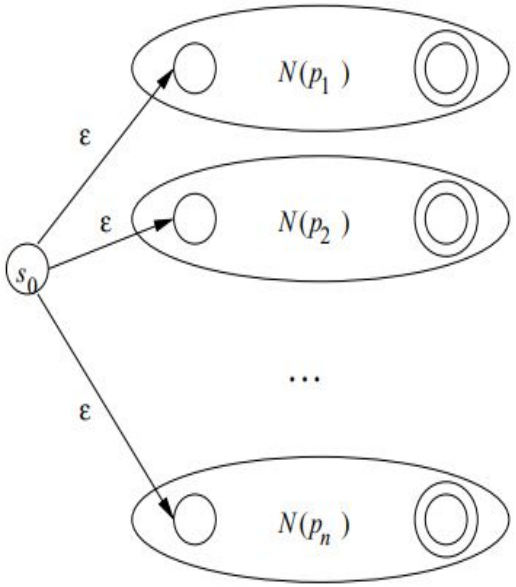


Figure 3.50: An NFA constructed from a Lex program

Steps to construct automaton

Step 1: Convert each regular expression into NFA either by Thompson's subset construction or Direct Method.

Step 2: Combine all NFAs into one by introducing new start state with s-transitions to each of start states of NFAs N_i for pattern P_i. Step 2 is needed as the objective is to construct single automaton to recognize lexemes that matches with any of the patterns.

a	{ action A_1 for pattern p_1 }
abb	{ action A_2 for pattern p_2 }
$\mathbf{a}^*\mathbf{b}^+$	{ action A_3 for pattern p_3 }

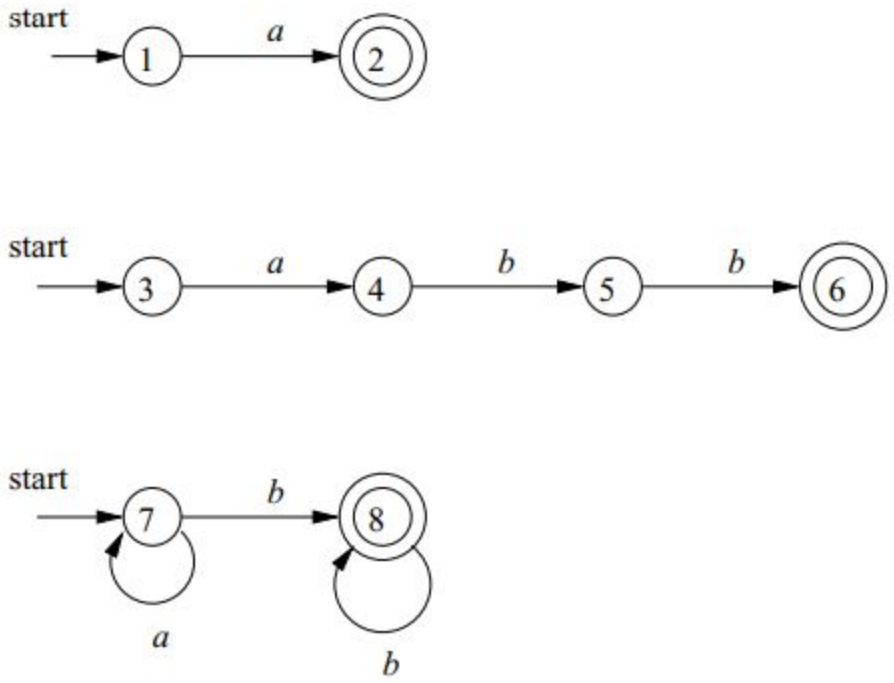


Figure 3.51: NFA's for **a**, **abb**, and $\mathbf{a}^*\mathbf{b}^+$

- For string *abb,* pattern P_2 and pattern p_3 matches.
- But the pattern P_2 will be taken into account as it was listed first in lex program.
- For string *aabbb* $\cdots$, matches pattern p_3 as it has many prefixes.

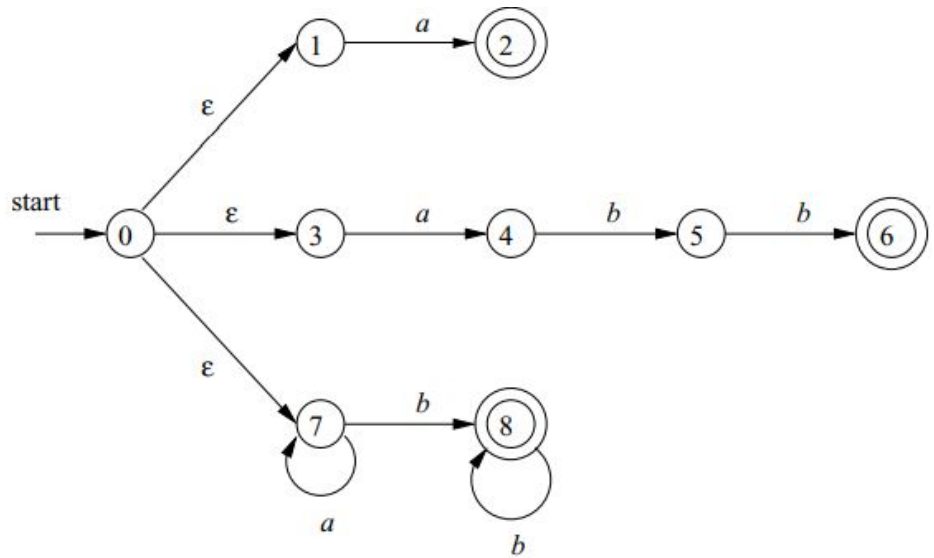


Figure 3.52: Combined NFA

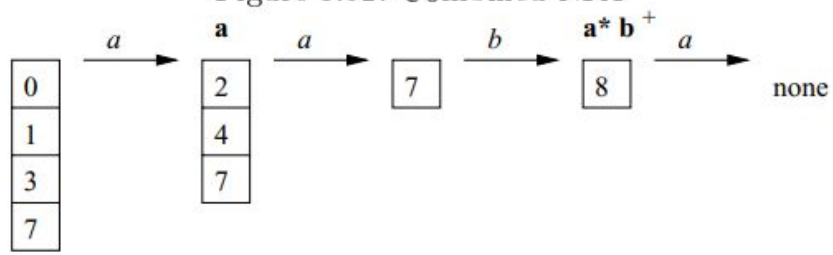


Figure 3.53: Sequence of sets of states entered when processing input *aaba*

Example 3.13 : In Fortran and some other languages, keywords are not reserved. That situation creates problems, such as a statement

```
IF(I,J) = 3
```

where `IF` is the name of an array, not a keyword. This statement contrasts with statements of the form

```
IF( condition ) THEN ...
```

where `IF` is a keyword. Fortunately, we can be sure that the keyword `IF` is always followed by a left parenthesis, some text — the condition — that may contain parentheses, a right parenthesis and a letter. Thus, we could write a `Lex` rule for the keyword `IF` like:

```
IF / \( .* \) {letter}
```

- This rule says that the pattern the lexeme matches is just the two letters IF.
- The slash says that additional pattern follows but does not match the lexeme. In this pattern,
- the first character is the left parentheses. Since that character is a Lex metasymbol, it must be preceded by a backslash to indicate that it has its literal meaning.
- The dot and star match \any string without a newline." Note that the dot is a Lex metasymbol meaning \any character except newline."
- It is followed by a right parenthesis, again with a backslash to give that character its literal meaning.

- The additional pattern is followed by the symbol letter, which is a regular definition representing the character class of all letters.
- Note that in order for this pattern to be foolproof, we must preprocess the input to delete whitespace.
- We have in the pattern neither provision for whitespace, nor can we deal with the possibility that the condition extends over lines, since the dot will not match a newline character.
- For instance, suppose this pattern is asked to match a prex of input: IF(A<(B+C)*D)THEN...

- the first two characters match IF, the next character matches \(, the next nine characters match .*, and the next two match \) and letter.
- Note the fact that the first right parenthesis (after C) is not followed by a letter is irrelevant; we only need to find some way of matching the input to the pattern. We conclude that the letters IF constitute the lexeme, and they are an instance of token if.